```
from numpy import *
from ODESolver import RungeKutta4

def rhs(u, t):
    R = 1
    return alpha*u*(1 - u/R)
```

$$\frac{du}{dt} = \alpha\, u(1-u)$$

$$u(0) = 0.1$$

$$R = 1$$

$$\alpha = 0.2$$

TEXTS IN COMPUTATIONAL SCIENCE AND ENGINEERING 6

Hans Petter Langtangen

# A Primer on Scientific Programming with Python



Springer

## *Editorial Policy*

§1. Textbooks on topics in the field of computational science and engineering will be considered. They should be written for courses in CSE education. Both graduate and undergraduate textbooks will be published in TCSE. Multidisciplinary topics and multidisciplinary teams of authors are especially welcome.

§2. Format: Only works in English will be considered. They should be submitted in camera-ready form according to Springer-Verlag's specifications.
Electronic material can be included if appropriate. Please contact the publisher.
Technical instructions and/or TEXmacros are available via
http://www.springer.com/authors/book+authors?SGWID=0-154102-12-417900-0

§3. Those considering a book which might be suitable for the series are strongly advised to contact the publisher or the series editors at an early stage.

## *General Remarks*

TCSE books are printed by photo-offset from the master-copy delivered in camera-ready form by the authors. For this purpose Springer-Verlag provides technical instructions for the preparation of manuscripts. See also *Editorial Policy*.

Careful preparation of manuscripts will help keep production time short and ensure a satisfactory appearance of the finished book.

The following terms and conditions hold:

Regarding free copies and royalties, the standard terms for Springer mathematics monographs and textbooks hold. Please write to martin.peters@springer.com for details.

Authors are entitled to purchase further copies of their book and other Springer books for their personal use, at a discount of 33,3% directly from Springer-Verlag.

# *Series Editors*

Timothy J. Barth
NASA Ames Research Center
NAS Division
Moffertt Field, CA 94035, USA
e-mail: barth@nas.nasa.gov

Michael Griebel
Institut für Numerische Simulation
der Universität Bonn
Wagelerstr. 6
53115 Bonn, Germany
e-mail: griebel@ins.uni-bonn.de

David E. Keyes
Department of Applied Physics
and Applied Mathematics
Columbia University
200 S. W. Mudd Building
500 W. 120th Street
New York, NY 10027, USA
e-mail: david.keyes@columbia.edu

Risto M. Nieminen
Laboratory of Physics
Helsinki University of Technology
02150 Espoo, Finland
e-mail: rni@fyslab.hut.fi

Dirk Roose
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven-Heverlee, Belgium
e-mail: dirk.roose@cs.kuleuven.ac.be

Tamar Schlick
Department of Chemistry
Courant Institute of Mathematical
Sciences
New York University
and Howard Hughes Medical Insitute
251 Mercer Street
New York, NY 10012, USA
e-mail: schlick@nyu.edu

Editor at Springer: Martin Peters
Springer-Verlag, Mathematics Editorial IV
Tiergartenstrasse 17
D-69121 Heidelberg, Germany
Tel.: *49 (6221) 487-8185
Fax: *49 (6221) 487-8355
e-mail: martin.peters@springer.com

# Texts in Computational Science and Engineering

1. H.P. Langtangen, *Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming. 2nd Edition
2. A. Quarteroni, F. Saleri, *Scientific Computing with MATLAB and Octave*. 2nd Edition
3. H.P. Langtangen, *Python Scripting for Computational Science*. 3rd Edition
4. H. Gardner, G. Manduchi, *Design Patterns for e-Science*.
5. M. Griebel, S. Knapek, G. Zumbusch, *Numerical Simulation in Molecular Dynamics*.
6. H.P. Langtangen, *A Primer on Scientific Programming with Python*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:*
`www.springer.com/series/5151`

# Monographs in Computational Science and Engineering

1. J. Sundnes, G.T. Lines, X. Cai, B.F. Nielsen, K.-A. Mardal, A. Tveito, *Computing the Electrical Activity in the Heart*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:*
`www.springer.com/series/7417`

# Lecture Notes in Computational Science and Engineering

1. D. Funaro, *Spectral Elements for Transport-Dominated Equations.*
2. H.P. Langtangen, *Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming.
3. W. Hackbusch, G. Wittum (eds.), *Multigrid Methods V.*
4. P. Deuflhard, J. Hermans, B. Leimkuhler, A.E. Mark, S. Reich, R.D. Skeel (eds.), *Computational Molecular Dynamics: Challenges, Methods, Ideas.*
5. D. Kröner, M. Ohlberger, C. Rohde (eds.), *An Introduction to Recent Developments in Theory and Numerics for Conservation Laws.*

6. S. Turek, *Efficient Solvers for Incompressible Flow Problems.* An Algorithmic and Computational Approach.
7. R. von Schwerin, *Multi Body System SIMulation.* Numerical Methods, Algorithms, and Software.
8. H.-J. Bungartz, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.*
9. T.J. Barth, H. Deconinck (eds.), *High-Order Methods for Computational Physics.*
10. H.P. Langtangen, A.M. Bruaset, E. Quak (eds.), *Advances in Software Tools for Scientific Computing.*
11. B. Cockburn, G.E. Karniadakis, C.-W. Shu (eds.), *Discontinuous Galerkin Methods.* Theory, Computation and Applications.
12. U. van Rienen, *Numerical Methods in Computational Electrodynamics.* Linear Systems in Practical Applications.
13. B. Engquist, L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid.*
14. E. Dick, K. Riemslagh, J. Vierendeels (eds.), *Multigrid Methods VI.*
15. A. Frommer, T. Lippert, B. Medeke, K. Schilling (eds.), *Numerical Challenges in Lattice Quantum Chromodynamics.*
16. J. Lang, *Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems.* Theory, Algorithm, and Applications.
17. B.I. Wohlmuth, *Discretization Methods and Iterative Solvers Based on Domain Decomposition.*
18. U. van Rienen, M. Günther, D. Hecht (eds.), *Scientific Computing in Electrical Engineering.*
19. I. Babuška, P.G. Ciarlet, T. Miyoshi (eds.), *Mathematical Modeling and Numerical Simulation in Continuum Mechanics.*
20. T.J. Barth, T. Chan, R. Haimes (eds.), *Multiscale and Multiresolution Methods.* Theory and Applications.
21. M. Breuer, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.*
22. K. Urban, *Wavelets in Numerical Simulation.* Problem Adapted Construction and Applications.
23. L.F. Pavarino, A. Toselli (eds.), *Recent Developments in Domain Decomposition Methods.*
24. T. Schlick, H.H. Gan (eds.), *Computational Methods for Macromolecules: Challenges and Applications.*
25. T.J. Barth, H. Deconinck (eds.), *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*.
26. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations*.
27. S. Müller, *Adaptive Multiscale Schemes for Conservation Laws*.
28. C. Carstensen, S. Funken, W. Hackbusch, R.H.W. Hoppe, P. Monk (eds.), *Computational Electromagnetics*.
29. M.A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*.
30. T. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (eds.), *Large-Scale PDE-Constrained Optimization*.
31. M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne (eds.), *Topics in Computational Wave Propagation*. Direct and Inverse Problems.
32. H. Emmerich, B. Nestler, M. Schreckenberg (eds.), *Interface and Transport Dynamics.* Computational Modelling.
33. H.P. Langtangen, A. Tveito (eds.), *Advanced Topics in Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming.

34. V. John, *Large Eddy Simulation of Turbulent Incompressible Flows.* Analytical and Numerical Results for a Class of LES Models.

35. E. Bänsch (ed.), *Challenges in Scientific Computing – CISC 2002.*

36. B.N. Khoromskij, G. Wittum, *Numerical Solution of Elliptic Differential Equations by Reduction to the Interface.*

37. A. Iske, *Multiresolution Methods in Scattered Data Modelling.*

38. S.-I. Niculescu, K. Gu (eds.), *Advances in Time-Delay Systems.*

39. S. Attinger, P. Koumoutsakos (eds.), *Multiscale Modelling and Simulation.*

40. R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Wildlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering.*

41. T. Plewa, T. Linde, V.G. Weirs (eds.), *Adaptive Mesh Refinement – Theory and Applications.*

42. A. Schmidt, K.G. Siebert, *Design of Adaptive Finite Element Software.* The Finite Element Toolbox ALBERTA.

43. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations II.*

44. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Methods in Science and Engineering.*

45. P. Benner, V. Mehrmann, D.C. Sorensen (eds.), *Dimension Reduction of Large-Scale Systems.*

46. D. Kressner, *Numerical Methods for General and Structured Eigenvalue Problems.*

47. A. Boriçi, A. Frommer, B. Joó, A. Kennedy, B. Pendleton (eds.), *QCD and Numerical Analysis III.*

48. F. Graziani (ed.), *Computational Methods in Transport.*

49. B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. Schütte, R. Skeel (eds.), *New Algorithms for Macromolecular Simulation.*

50. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations.*

51. A.M. Bruaset, A. Tveito (eds.), *Numerical Solution of Partial Differential Equations on Parallel Computers.*

52. K.H. Hoffmann, A. Meyer (eds.), *Parallel Algorithms and Cluster Computing.*

53. H.-J. Bungartz, M. Schäfer (eds.), *Fluid–Structure Interaction.*

54. J. Behrens, *Adaptive Atmospheric Modeling.*

55. O. Widlund, D. Keyes (eds.), *Domain Decomposition Methods in Science and Engineering XVI.*

56. S. Kassinos, C. Langer, G. Iaccarino, P. Moin (eds.), *Complex Effects in Large Eddy Simulations.*

57. M. Griebel, M. A Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations III.*

58. A.N. Gorban, B. Kégl, D.C. Wunsch, A. Zinovyev (eds.), *Principal Manifolds for Data Visualization and Dimension Reduction.*

59. H. Ammari (ed.), *Modeling and Computations in Electromagnetics: A Volume Dedicated to Jean-Claude Nédélec.*

60. U. Langer, M. Discacciati, D. Keyes, O. Widlund, W. Zulehner (eds.), *Domain Decomposition Methods in Science and Engineering XVII.*

61. T. Mathew, *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations.*

62. F. Graziani (ed.), *Computational Methods in Transport: Verification and Validation.*

63. M. Bebendorf, *Hierarchical Matrices.* A Means to Efficiently Solve Elliptic Boundary Value Problems.
64. C.H. Bischof, H.M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.), *Advances in Automatic Differentiation.*
65. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations IV*.
66. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Modeling and Simulation in Science.*
67. I.H. Tuncer, Ü. Gülcat, D.R. Emerson, K. Matsuno (eds.), *Parallel Computational Fluid Dynamics.*
68. S. Yip, T. Diaz de la Rubia (eds.), *Scientific Modeling and Simulations*.
69. A. Hegarty, N. Kopteva, E. O'Riordan, M. Stynes (eds.), *BAIL 2008 – Boundary and Interior Layers.*
70. M. Bercovier, M.J. Gander, R. Kornhuber, O. Widlund (eds.), *Domain Decomposition Methods in Science and Engineering XVIII*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:* www.springer.com/series/3527

Texts in Computational Science
and Engineering

6

Hans Petter Langtangen

# A Primer on Scientific Programming with Python

Springer

Hans Petter Langtangen

Simula Research Laboratory
Martin Linges vei 17
1325 Lysaker, Fornebu
Norway
hpl@simula.no

On leave from:

Department of Informatics
University of Oslo
P.O. Box 1080 Blindern
0316 Oslo, Norway
http://folk.uio.no/hpl

ISSN 1611-0994
ISBN 978-3-642-02474-0 e-ISBN 978-3-642-02475-7
DOI 10.1007/978-3-642-02475-7
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2009931367

Mathematics Subject Classification (2000): 26-01, 34A05, 34A30, 34A34, 39-01, 40-01, 65D15, 65D25, 65D30, 68-01, 68N01, 68N19, 68N30, 70-01, 92D25, 97-04, 97U50



*Production:* le-tex publishing services GmbH, Leipzig, Germany
*Cover design:* deblik, Berlin

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

The aim of this book is to teach computer programming using examples from mathematics and the natural sciences. We have chosen to use the Python programming language because it combines remarkable power with very clean, simple, and compact syntax. Python is easy to learn and very well suited for an introduction to computer programming. Python is also quite similar to Matlab and a good language for doing mathematical computing. It is easy to combine Python with compiled languages, like Fortran, C, and C++, which are widely used languages for scientific computations. A seamless integration of Python with Java is offered by a special version of Python called Jython.

The examples in this book integrate programming with applications to mathematics, physics, biology, and finance. The reader is expected to have knowledge of basic one-variable calculus as taught in mathematics-intensive programs in high schools. It is certainly an advantage to take a university calculus course in parallel, preferably containing both classical and numerical aspects of calculus. Although not strictly required, a background in high school physics makes many of the examples more meaningful.

Many introductory programming books are quite compact and focus on listing functionality of a programming language. However, learning to program is learning how to *think* as a programmer. This book has its main focus on the thinking process, or equivalently: programming as a problem solving technique. That is why most of the pages are devoted to case studies in programming, where we define a problem and explain how to create the corresponding program. New constructions and programming styles (what we could call theory) is also usually introduced via examples. Special attention is paid to verification of programs and to finding errors. These topics are very demanding for mathematical software, because we have approximation errors possibly mixed with programming errors.

By studying the many examples in the book, I hope readers will learn how to think right and thereby write programs in a quicker and more reliable way. Remember, nobody can learn programming by just reading – one has to solve a large amount of exercises hands on. Therefore, the book is full of exercises of various types: modifications of existing examples, completely new problems, or debugging of given programs.

To work with this book, you need to install Python version 2.6. In Chapter 4 and later chapters, you also need the NumPy and SciTools packages. To make curve plots with SciTools, you must have a plotting program, for example, Gnuplot or Matplotlib. There is a web page associated with this book, `http://www.simula.no/intro-programming`, which lists the software you need and explains briefly how to install it. On this page, you will also find all the files associated with the program examples in this book. Download `book-examples.tar.gz`, store this file in some folder of your choice, and unpack it using WinZip on Windows or the command `tar xzf book-examples.tar.gz` on Linux and Mac. This unpacking yields a folder `src` with subfolders for the various chapters in the book.

*Contents.* Chapter 1 introduces variables, objects, modules, and text formatting through examples concerning evaluation of mathematical formulas. Chapter 2 presents fundamental elements of programming: loops, lists, and functions. This is a comprehensive and important chapter that should be digested before proceeding. How to read data into programs and deal with errors in input are the subjects of Chapter 3. Many of the examples in the first three chapters are strongly related. Typically, formulas from the first chapter are encapsulated in functions in the second chapter, and in the third chapter the input to the functions are fetched from the command line or from a question-answer dialog with the user, and validity of the data is checked. Chapter 4 introduces arrays and array computing (including vectorization) and how this is used for plotting $y = f(x)$ curves. After the first four chapters, the reader should have enough knowledge of programming to solve mathematical problems by "Matlab-style" programming.

Chapter 5 introduces mathematical modeling, using sequences and difference equations. We also treat sound as a sequence. No new programming concepts are introduced in this chapter, the aim being to consolidate the programming knowledge and apply it to mathematical problems.

Chapter 6 explains how to work with files and text data. Class programming, including user-defined types for mathematical computations (with overloaded operators), is introduced in Chapter 7. Chapter 8 deals with random numbers and statistical computing with applications to games and random walk. Object-oriented programming (class hierarchies and inheritance) is the subject of Chapter 9. The

key examples here deal with building toolkits for graphics and for numerical differentiation, integration, and solution of ordinary differential equations.

Appendix A deals with functions on a mesh, numerical differentiation, and numerical integration. The next appendix gives an introduction to numerical solution of ordinary differential equations. These two appendices provide the theoretical background on numerical methods that are much in use in Chapters 7 and 9. Moreover, the appendices exemplify basic programming from the first four chapters.

Appendix C shows how a complete project in physics can be solved by mathematical modeling, numerical methods, and programming elements from the first four chapters. This project is a good example on problem solving in computational science, where it is necessary to integrate physics, mathematics, numerics, and computer science.

Appendix D is devoted to the art of debugging, and in fact problem solving in general, while Appendix E deals with various more advanced technical topics.

Most of the examples and exercises in this book are quite compact and limited. However, many of the exercises are related, and together they form larger projects in science, for example on Fourier Series (1.13, 2.39, 3.17, 3.18, 3.19, 4.20), Taylor series (2.38, 4.16, 4.18, 5.16, 5.17, 7.31), falling objects (7.25, 9.30, 7.26, 9.31, 9.32, 9.34), oscillatory population growth (5.21, 5.22, 6.28, 7.41, 7.42), visualization of web data (6.22, 6.23, 6.24, 6.25), graphics and animation (9.36, 9.37, 9.38, 9.39), optimization and finance (5.23, 8.42, 8.43), statistics and probability (3.25, 3.26, 3.27, 8.19, 8.20, 8.21), random walk and statistical physics (8.33, 8.34, 8.35, 8.36, 8.37, 8.38, 8.39, 8.40), noisy data analysis (8.44, 8.45, 8.46, 8.47, 9.40), numerical methods (5.12, 7.9, 7.22, 9.15, 9.16, 9.26, 9.27, 9.28), building a calculus calculator (9.41, 9.42, 9.43, 9.44), and creating a toolkit for simulating oscillatory systems (9.45–9.52).

Chapters 1–9 and Appendix C form the core of an introductory first-semester course on scientific programming (INF1100) at the University of Oslo. Normally, each chapter is suited for a $2 \times 45$ min lecture, but Chapters 2 and 7 are challenging, and each of them have consumed two lectures in the course.

*Acknowledgments.* First, I want to express my thanks to Aslak Tveito for his enthusiastic role in the initiation of this book project and for writing Appendices A and B about numerical methods. Without Aslak there would be no book. Another key contributor is Ilmar Wilbers. His extensive efforts with assisting the book project and teaching the associated course (INF1100) at the University of Oslo are greatly appreciated. Without Ilmar and his solutions to numerous technical problems the book would never have been completed. Johannes H. Ring also deserves a special acknowledgment for the development of the Easyviz

graphics tool, which is much used throughout this book, and for his careful maintenance and support of software associated with this book.

Several people have helped to make substantial improvements of the text, the exercises, and the associated software infrastructure. The author is thankful to Ingrid Eide, Tobias Vidarssønn Langhoff, Mathias Nedrebø, Arve Knudsen, Marit Sandstad, Lars Storjord, Fredrik Heffer Valdmanis, and Torkil Vederhus for their contributions. Hakon Adler is greatly acknowledged for his careful reading of various versions of the manuscript. The professors Fred Espen Bent, Ørnulf Borgan, Geir Dahl, Knut Mørken, and Geir Pedersen have contributed with many exciting exercises from various application fields. Great thanks also go to Jan Olav Langseth for creating the cover image.

This book and the associated course are parts of a comprehensive reform at the University of Oslo, called *Computers in Science Education*. The goal of the reform is to integrate computer programming and simulation in all bachelor courses in natural science where mathematical models are used. The present book lays the foundation for the modern computerized problem solving technique to be applied in later courses. It has been extremely inspiring to work with the driving forces behind this reform, in particular the professors Morten Hjorth–Jensen, Anders Malthe–Sørenssen, Knut Mørken, and Arnt Inge Vistnes.

The excellent assistance from the Springer and Le-TeX teams, consisting of Martin Peters, Thanh-Ha Le Thi, Ruth Allewelt, Peggy Glauch-Ruge, Nadja Kroke, and Thomas Schmidt, is highly appreciated and ensured a smooth and rapid production of this book.

*Oslo, May 2009* *Hans Petter Langtangen*

# Contents

# List of Exercises

# 1 Computing with Formulas

Our first examples on computer programming involve programs that evaluate mathematical formulas. You will learn how to write and run a Python program, how to work with variables, how to compute with mathematical functions such as $e^x$ and $\sin x$, and how to use Python for interactive calculations.

We assume that you are somewhat familiar with computers so that you know what files and folders[1] are, how you move between folders, how you change file and folder names, and how you write text and save it in a file.

All the program examples associated with this chapter can be found as files in the folder `src/formulas`. We refer to the preface for how to download the folder tree `src` containing all the program files for this book.

## 1.1 The First Programming Encounter: A Formula

The first formula we shall consider concerns the vertical motion of a ball thrown up in the air. From Newton's second law of motion one can set up a mathematical model for the motion of the ball and find that the vertical position of the ball, called $y$, varies with time $t$ according to the following formula[2]:

$$y(t) = v_0 t - \frac{1}{2} g t^2 \,. \tag{1.1}$$

[1] Another frequent word for folder is directory.

[2] This formula neglects air resistance, which is usually small unless $v_0$ is large – see Exercise 1.10.

Here, $v_0$ is the initial velocity of the ball, $g$ is the acceleration of gravity, and $t$ is time. Observe that the $y$ axis is chosen such that the ball starts at $y = 0$ when $t = 0$.

To get an overview of the time it takes for the ball to move upwards and return to $y = 0$ again, we can look for solutions to the equation $y = 0$:

$$v_0 t - \frac{1}{2}gt^2 = t(v_0 - \frac{1}{2}gt) = 0 \quad : \quad t = 0 \text{ or } t = 2v_0/g\,.$$

That is, the ball returns after $2v_0/g$ seconds, and it is therefore reasonable to restrict the interest of (1.1) to $t \in [0, 2v_0/g]$.

### 1.1.1 Using a Program as a Calculator

Our first program will evaluate (1.1) for a specific choice of $v_0$, $g$, and $t$. Choosing $v_0 = 5$ m/s and $g = 9.81$ m/s$^2$ makes the ball come back after $t = 2v_0/g \approx 1$ s. This means that we are basically interested in the time interval $[0, 1]$. Say we want to compute the height of the ball at time $t = 0.6$ s. From (1.1) we have

$$y = 5 \cdot 0.6 - \frac{1}{2} \cdot 9.81 \cdot 0.6^2$$

This arithmetic expression can be evaluated and its value can be printed by a very simple one-line Python program:

```
print 5*0.6 - 0.5*9.81*0.6**2
```

The four standard arithmetic operators are written as `+`, `-`, `*`, and `/` in Python and most other computer languages. The exponentiation employs a double asterisk notation in Python, e.g., $0.6^2$ is written as `0.6**2`.

Our task now is to create the program and run it, and this will be described next.

### 1.1.2 About Programs and Programming

A computer program is just a sequence of instructions to the computer, written in a computer language. Most computer languages look somewhat similar to English, but they are very much simpler. The number of words and associated instructions is very limited, so to perform a complicated operation we must combine a large number of different types of instructions. The program text, containing the sequence of instructions, is stored in one or more files. The computer can only do exactly what the program tells the computer to do.

Another perception of the word "program" is a file that can be run ("double-clicked") to perform a task. Sometimes this is a file with textual instructions (which is the case with Python), and sometimes this file is a translation of all the program text to a more efficient and computer-friendly language that is quite difficult to read for a human. All the programs in this chapter consist of short text stored in a single file. Other programs that you have used frequently, for instance Firefox or Internet Explorer for reading web pages, consist of program text distributed over a large number of files, written by a large number of people over many years. One single file contains the machine-efficient translation of the whole program, and this is normally the file that you "double-click" on when starting the program. In general, the word "program" means either this single file or the collection of files with textual instructions.

Programming is obviously about writing programs, but this process is more than writing the correct instructions in a file. First, we must understand how a problem can be solved by giving a sequence of instructions to the computer. This is usually the most difficult thing with programming. Second, we must express this sequence of instructions correctly in a computer language and store the corresponding text in a file (the program). Third, we must run the program, check the validity of the results, and usually enter a fourth phase where errors in the program must be found and corrected. Mastering this process requires a lot of training, which implies making a large number of programs (exercises in this book, for instance) and getting the programs to work.

### 1.1.3 Tools for Writing Programs

Since programs consist of plain text, we need to write this text with the help of another program that can store the text in a file. You have most likely extensive experience with writing text on a computer, but for writing your own programs you need special programs, called *editors*, which preserve exactly the characters you type. The widespread word processors, Microsoft Word being a primary example[3], are aimed at producing nice-looking reports. These programs *format* the text and are *not* good tools for writing your own programs, even though they can save the document in a pure text format. Spaces are often important in Python programs, and *editors* for plain text give you complete control of the spaces and all other characters in the program file.

[3] Other examples are OpenOffice, TextEdit, iWork Pages, and BBEdit. Chapter 6.1.3 gives some insight into why such programs are not suitable for writing your own Python programs.

Emacs, XEmacs, and Vim are popular editors for writing programs on Linux or Unix systems, including Mac[4] computers. On Windows we recommend Notepad++ or the Window versions of Emacs or Vim. None of these programs are part of a standard Windows installation.

A special editor for Python programs comes with the Python software. This editor is called Idle and is usually installed under the name `idle` (or `idle-python`) on Linux/Unix and Mac. On Windows, it is reachable from the Python entry in the Start menu. Idle has a gentle learning curve, but is mainly restricted to writing Python programs. Completely general editors, such as Emacs and Vim, have a steeper learning curve and can be used for any text files, including reports in student projects.

### 1.1.4 Using Idle to Write the Program

Let us explain in detail how we can use Idle to write our one-line program from Chapter 1.1.1. Idle may not become your favorite editor for writing Python programs, yet we recommend to follow the steps below to get in touch with Idle and try it out. You can simply replace the Idle instructions by similar actions in your favorite editor, Emacs for instance.

First, create a folder where your Python programs can be located. Here we choose a folder name `py1st` under your home folder (note that the third character is the number 1, not the letter l – the name reflects your 1st try of Python). To write and run Python programs, you will need a terminal window on Linux/Unix or Mac, sometimes called a console window, or a DOS window on Windows. Launch such a window and use the `cd` (change directory) command to move to the `py1st` folder. If you have not made the folder with a graphical file & folder manager you must create the folder by the command `mkdir py1st` (`mkdir` stands for make directory).

The next step is to start Idle. This can be done by writing `idle&` (Linux) or `start idle` (Windows) in the terminal window. Alternatively, you can launch Idle from the Start menu on Windows. Figure 1.1 displays a terminal window where we create the folder, move to the folder, and start Idle[5].

If a window now appears on the screen, with "Python Shell" in the title bar of the window, go to its File menu and choose New Window.

[4] On Mac, you may want to download a more "Mac-like" editor such as the Really Simple Text program.

[5] The ampersand after `idle` is Linux specific. On Windows you have to write `start idle` instead. The ampersand postfix or the `start` prefix makes it possible to continue with other commands in the terminal window while the program, here Idle, is running. This is important for program testing where we often do a lot of edit-and-run cycles, which means that we frequently switch between the editor and the terminal window.




**Fig. 1.1** A terminal window on a Linux/Unix/Mac machine where we create a folder (`mkdir`), move to the folder (`cd`), and start Idle.

The window that now pops up is the Idle editor (having the window name "Untitled"). Move the cursor inside this window and write the line

```
print 5*0.6 - 0.5*9.81*0.6**2
```

followed by pressing the Return key. The Idle window looks as in Figure 1.2.

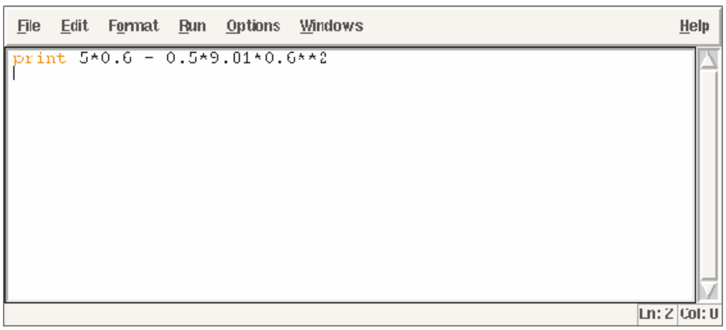



**Fig. 1.2** An Idle editor window containing our first one-line program.

Your program is now in the Idle editor, but before you can run it, the program text must be saved in a file. Choose File and then Save As. As usual, such a command launches a new window where you can fill in the name of the file where the program is to be stored. And as always, you must first check that you are in the right folder, or directory which is Idle's word for the same thing. The upper line in the file dialog window contains the folder name. Clicking on the bar to the right (after the directory/folder name), gives a possibility to move upwards in the folder hierarchy, and clicking on the folder icon to the right of the bar, moves just one folder upwards. To go down in the folder tree, you simply double-click a folder icon in the main window of this dialog. You must now navigate to the `py1st` folder under your home folder. If you started Idle from the terminal window, there

is no need to navigate and change folder. Simply fill in the name of the program. Any name will do, but we suggest that you choose the name `ball_numbers.py` because this name is compatible with what we use later in this book. The file extension `.py` is common for Python programs, but not strictly required[6].

Press the Save button and move back to the terminal window. Make sure you have a new file `ball_numbers.py` here, by running the command `ls` (on Linux/Unix and Mac) or `dir` (on Windows). The output should be a text containing the name of the program file. You can now jump to the paragraph "How to Run the Program", but it might be a good idea to read the warning below first.

*Warning About Typing Program Text.* Even though a program is just a text, there is one major difference between a text in a program and a text intended to be read by a human. When a human reads a text, she or he is able to understand the message of the text even if the text is not perfectly precise or if there are grammar errors. If our one-line program was expressed as

```
write 5*0.6 - 0.5*9.81*0.6^2
```

most humans would interpret `write` and `print` as the same thing, and many would also interpret `6^2` as $6^2$. In the Python language, however, `write` is a grammar error and `6^2` means an operation very different from the exponentiation `6**2`. Our communication with a computer through a program must be perfectly precise without a single grammar error[7]. The computer will only do exactly what we tell it to do. Any error in the program, however small, may affect the program. There is a chance that we will never notice it, but most often an error causes the program to stop or produce wrong results. The conclusion is that computers have a much more pedantic attitude to language than what (most) humans have.

Now you understand why any program text must be carefully typed, paying attention to the correctness of every character. If you try out program texts from this book, make sure that you type them in *exactly as you see them* in the book. Blanks, for instance, are often important in Python, so it is a good habit to always count them and type them in correctly. Any attempt not to follow this advice will cause you frustrations, sweat, and maybe even tears.

[6] Some editors, like Emacs, have many features that make it easier to write Python programs, but these features will not be automatically available unless the program file has a `.py` extension.

[7] "Programming demands significantly higher standard of accuracy. Things don't simply have to make sense to another human being, they must make sense to a computer." –Donald Knuth [4, p. 18], computer scientist, 1938-.

### 1.1.5 How to Run the Program

The one-line program above is stored in a file with name `ball_numbers.py`. To run the program, you need to be in a terminal window and in the folder where the `ball_numbers.py` file resides. The program is run by writing the command `python ball_numbers.py` in the terminal window[8]:

Terminal

```
Unix/DOS> python ball_numbers.py
1.2342
```

The program immediately responds with printing the result of its calculation, and the output appears on the next line in the terminal window. In this book we use the prompt `Unix/DOS>` to indicate a command line in a Linux, Unix, Mac, or DOS terminal window (a command line means that we can run Unix or DOS commands, such as `cd` and `python`). On your machine, you will likely see a different prompt. Figure 1.3 shows what the whole terminal window may look like after having run the program.

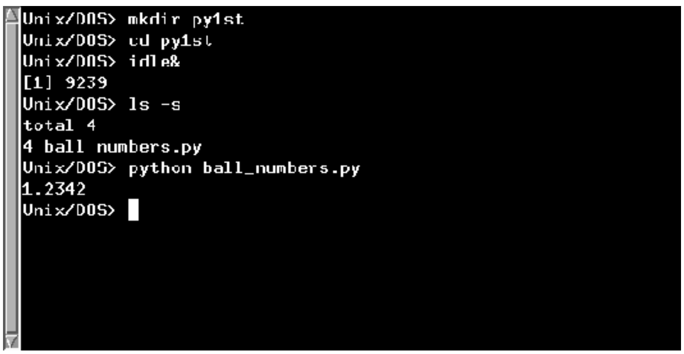



**Fig. 1.3** A terminal window on a Linux/Unix/Mac machine where we run our first one-line Python program.

From your previous experience with computers you are probably used to double-click on icons to run programs. Python programs can also be run that way, but programmers usually find it more convenient to run programs by typing commands in a terminal window. Why this is so will be evident later when you have more programming experience. For now, simply accept that you are going to be a programmer, and that commands in a terminal window is an efficient way to work with the computer.

Suppose you want to evaluate (1.1) for $v_0 = 1$ and $t = 0.1$. This is easy: move the cursor to the Idle editor window, edit the program text to

[8] There are other ways of running Python programs, as explained in Appendix E.1.

```
print 1*0.1 - 0.5*9.81*0.1**2
```

Save the file, move back to the terminal window and run the program as before:

Terminal
```
Unix/DOS> python ball_numbers.py
0.05095
```

We see that the result of the calculation has changed, as expected.

### 1.1.6 Verifying the Result

We should *always* carefully control that the output of a computer program is correct. You will experience that in most of the cases, at least until you are an experienced programmer, the output is wrong, and you have to search for errors. In the present application we can simply use a calculator to control the program. Setting $t = 0.6$ and $v_0 = 5$ in the formula, the calculator confirms that 1.2342 is the correct solution to our mathematical problem.

### 1.1.7 Using Variables

When we want to evaluate $y(t)$ for many values of $t$, we must modify the $t$ value at two places in our program. Changing another parameter, like $v_0$, is in principle straightforward, but in practice it is easy to modify the wrong number. Such modifications would be simpler to perform if we express our formula in terms of variables, i.e., symbols, rather than numerical values. Most programming languages, Python included, have variables similar to the concept of variables in mathematics. This means that we can define `v0`, `g`, `t`, and `y` as variables in the program, initialize the former three with numerical values, and combine these three variables to the desired right-hand side expression in (1.1), and assign the result to the variable `y`.

The alternative version of our program, where we use variables, may be written as this text:

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Figure 1.4 displays what the program looks like in the Idle editor window. Variables in Python are defined by setting a name (here `v0`, `g`, `t`, or `y`) equal to a numerical value or an expression involving already defined variables.

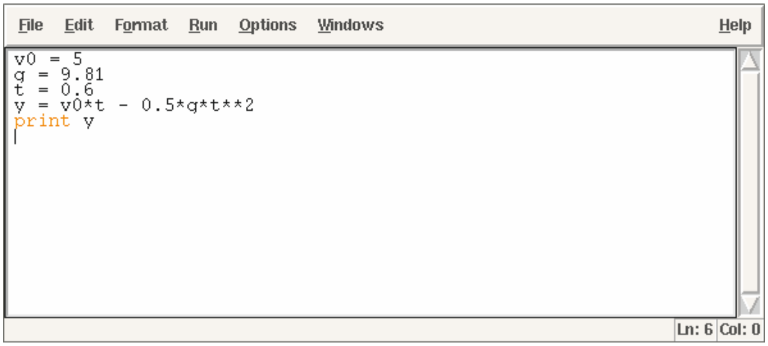



**Fig. 1.4** An Idle editor window containing a multi-line program with several variables.

Note that this second program is much easier to read because it is closer to the mathematical notation used in the formula (1.1). The program is also safer to modify, because we clearly see what each number is when there is a name associated with it. In particular, we can change `t` at one place only (the line `t = 0.6`) and not two as was required in the previous program.

We store the program text in a file `ball_variables.py`. Running the program,

Terminal
```
Unix/DOS> python ball_variables.py
```

results in the correct output `1.2342`.

### 1.1.8 Names of Variables

Introducing variables with descriptive names, close to those in the mathematical problem we are going to solve, is considered important for the readability and reliability (correctness) of the program. Variable names can contain any lower or upper case letter, the numbers from 0 to 9, and underscore, but the first character cannot be a number. Python distinguishes between upper and lower case, so `X` is always different from `x`. Here are a few examples on alternative variable names in the present example[9]:

```
initial_velocity = 5
acceleration_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = initial_velocity*TIME - \
                         0.5*acceleration_of_gravity*TIME**2
print VerticalPositionOfBall
```

[9] In this book we shall adopt the rule that variable names have lower case letters where words are separated by an underscore. The first two declared variables have this form.

With such long variables names, the code for evaluating the formula becomes so long that we have decided to break it into two lines. This is done by a backslash at the very end of the line (make sure there are no blanks after the backslash!).

We note that even if this latter version of the program contains variables that are defined precisely by their names, the program is harder to read than the one with variables `v0`, `g`, `t`, and `y0`.

The rule of thumb is to use the same variable names as those appearing in a precise mathematical description of the problem to be solved by the program. For all variables where there is no associated precise mathematical description and symbol, one must use *descriptive* variable names which explain the purpose of the variable. For example, if a problem description introduces the symbol $D$ for a force due to air resistance, one applies a variable `D` also in the program. However, if the problem description does not define any symbol for this force, one must apply a descriptive name, such as `air_resistance`, `resistance_force`, or `drag_force`.

### 1.1.9 Reserved Words in Python

Certain words are reserved in Python because they are used to build up the Python language. These reserved words cannot be used as variable names: `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `with`, `while`, and `yield`. You may, for instance, add an underscore at the end to turn a reserved word into a variable name. See Exercise 1.16 for examples on legal and illegal variable names.

### 1.1.10 Comments

Along with the program statements it is often informative to provide some comments in a natural human language to explain the idea behind the statements. Comments in Python start with the `#` character, and everything after this character on a line is ignored when the program is run. Here is an example of our program with explanatory comments:

```
# program for computing the height of a ball thrown up in the air
v0 = 5    # initial velocity
g = 9.81  # acceleration of gravity
t = 0.6   # time
y = v0*t - 0.5*g*t**2  # vertical position
print y
```

This program and the initial version on page 8 are identical when run on the computer, but for a human the latter is easier to understand because of the comments.

Good comments together with well-chosen variable names are necessary for any program longer than a few lines, because otherwise the program becomes difficult to understand, both for the programmer and others. It requires some practice to write really instructive comments. Never repeat with words what the program statements already clearly express. Use instead comments to provide important information that is not obvious from the code, for example, what mathematical variable names mean, what variables are used for, and general ideas that lie behind a forthcoming set of statements.

### 1.1.11 Formatting Text and Numbers

Instead of just printing the numerical value of `y` in our introductory program, we now want to write a more informative text, typically something like

```
At t=0.6 s, the height of the ball is 1.23 m.
```

where we also have control of the number of digits (here $y$ is accurate up to centimeters only).

Such output from the program is accomplished by a `print` statement where we use something often known as *printf* formatting[10]. For a newcomer to programming, the syntax of printf formatting may look awkward, but it is quite easy to learn and very convenient and flexible to work with. The sample output above is produced by this statement:

```
print 'At t=%g s, the height of the ball is %.2f m.' % (t, y)
```

Let us explain this line in detail. The `print` statement now prints a string: everything that is enclosed in quotes (either single: `'`, or double: `"`) denotes a string in Python. The string above is formatted using printf syntax. This means that the string has various "slots", starting with a percentage sign, here `%g` and `%.2f`, where variables in the program can be put in. We have two "slots" in the present case, and consequently two variables must be put into the slots. The relevant syntax is to list the variables inside standard parentheses after the string, separated from the string by a percentage sign. The first variable, `t`, goes into the first "slot". This "slot" has a format specification `%g`, where the percentage sign marks the slot and the following character, `g`, is a format specification. The `g` that a real number is to be written as compactly as possible. The next variable, `y`, goes into the second "slot". The format specification here is `.2f`, which means a real number written with two digits after comma. The `f` in the `.2f` format

[10] This formatting was originally introduced by a function `printf` in the C programming language.

stands for *float*, a short form for *floating-point number*, which is the term used for a real number on a computer.

For completeness we present the whole program, where text and numbers are mixed in the output:

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print 'At t=%g s, the height of the ball is %.2f m.' % (t, y)
```

You can find the program in the file `ball_output1.py` in the `src/formulas` folder.

There are many more ways to specify formats. For example, `e` writes a number in *scientific notation*, i.e., with a number between 1 and 10 followed by a power of 10, as in $1.2432 \cdot 10^{-3}$. On a computer such a number is written in the form `1.2432e-03`. Capital `E` in the exponent is also possible, just replace `e` by `E`, with the result `1.2432E-03`.

For *decimal notation* we use the letter `f`, as in `%f`, and the output number then appears with digits before and/or after a comma, e.g., `0.0012432` instead of `1.2432E-03`. With the `g` format, the output will use scientific notation for large or small numbers and decimal notation otherwise. This format is normally what gives most compact output of a real number. A lower case `g` leads to lower case `e` in scientific notation, while upper case `G` implies `E` instead of `e` in the exponent.

One can also specify the format as `10.4f` or `14.6E`, meaning in the first case that a float is written in decimal notation with four decimals in a field of width equal to 10 characters, and in the second case a float written in scientific notation with six decimals in a field of width 14 characters.

Here is a list of some important printf format specifications[11]:

```
%s                a string
%d                an integer
%0xd              an integer padded with x leading zeros
%f                decimal notation with six decimals
%e                compact scientific notation, e in the exponent
%E                compact scientific notation, E in the exponent
%g                compact decimal or scientific notation (with e)
%G                compact decimal or scientific notation (with E)
%xz               format z right-adjusted in a field of width x
%-xz              format z left-adjusted  in a field of width x
%.yz              format z with y decimals
%x.yz             format z with y decimals in a field of width x
%%                the percentage sign (%) itself
```

The program `printf_demo.py` exemplifies many of these formats.

We may try out some formats by writing more numbers to the screen in our program (the corresponding file is `ball_output2.py`):

[11] For a complete specification of the possible printf-style format strings, follow the link from the item "printf-style formatting" in the *index* of the Python Library Reference.

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print """
At t=%f s, a ball with
initial velocity v0=%.3E m/s
is located at the height %.2f m.
""" % (t, v0, y)
```

Observe here that we use a *triple-quoted* string, recognized by starting and ending with three single or double quotes: ■’ or `"""`. Triple-quoted strings are used for text that spans several lines.

In the `print` statement above, we write `t` in the `f` format, which by default implies six decimals; `v0` is written in the `.3E` format, which implies three decimals and the number spans as narrow field as possible; and `y` is written with two decimals in decimal notation in as narrow field as possible. The output becomes

Terminal

```
Unix/DOS> python ball_fmt2.py

At t=0.600000 s, a ball with
initial velocity v0=5.000E+00 m/s
is located at the height 1.23 m.
```

You should look at each number in the output and check the formatting in detail.

*The Newline Character.* We often want a computer program to write out text that spans several lines. In the last example we obtained such output by triple-quoted strings. We could also use ordinary single-quoted strings and a special character for indicating where line breaks should occur. This special character reads `\n`, i.e., a backslash followed by the letter `n`. The two `print` statements

```
print """y(t) is
the position of
our ball."""

print 'y(t) is\nthe position of\nour ball'
```

result in identical output:

```
y(t) is
the position of
our ball.
```

## 1.2 Computer Science Glossary

It is now time to pick up some important words that programmers use when they talk about programming: algorithm, application, assignment, blanks (whitespace), bug, code, code segment, code snippet,

debug, debugging, execute, executable, implement, implementation, input, library, operating system, output, statement, syntax, user, verify, and verification.
These words are frequently used in English in lots of contexts, yet they have a precise meaning in computer science.

*Program* and *code* are interchangeable terms. A code/program *segment* is a collection of consecutive statements from a program. Another term with similar meaning is *code snippet*. Many also use the word *application* in the same meaning as program and code. A related term is *source code*, which is the same as the text that constitutes the program. You find the source code of a program in one or more text files[12].

We talk about *running a program*, or equivalently *executing a program* or *executing a file*. The file we execute is the file in which the program text is stored. This file is often called an *executable* or an *application*. The program text may appear in many files, but the executable is just the single file that starts the whole program when we run that file. Running a file can be done in several ways, for instance, by double-clicking the file icon, by writing the filename in a terminal window, or by giving the filename to some program. This latter technique is what we have used so far in this book: we feed the filename to the program `python`. That is, we execute a Python program by executing another program `python`, which interprets the text in our Python program file.

The term *library* is widely used for a collection of generally useful program pieces that can be applied in many different contexts. Having access to good libraries means that you do not need to program code snippets that others have already programmed (most probable in a better way!). There are huge numbers of Python libraries. In Python terminology, the libraries are composed of *modules* and *packages*. Chapter 1.4 gives a first glimpse of the `math` module, which contains a set of standard mathematical functions for $\sin x$, $\cos x$, $\ln x$, $e^x$, $\sinh x$, $\sin^{-1} x$, etc. Later, you will meet many other useful modules. Packages are just collections of modules. The standard Python distribution comes with a large number of modules and packages, but you can download many more from the Internet, see in particular `www.python.org/pypi`. Very often, when you encounter a programming task that is likely to occur in many other contexts, you can find a Python module where the job is already done. To mention just one example, say you need to compute how many days there are between two dates. This is a non-trivial task that lots of other programmers must have faced, so it is not a big surprise that Python comes with a module `datetime` to do calculations with dates.

[12] Note that text files normally have the extension `.txt`, while program files have an extension related to the programming language, e.g., `.py` for Python programs. The content of a `.py` file is, nevertheless, plain text as in a `.txt` file.

The recipe for what the computer is supposed to do in a program is called *algorithm*. In the examples in the first couple of chapters in this book, the algorithms are so simple that we can hardly distinguish them from the program text itself, but later in the book we will carefully set up an algorithm before attempting to *implement* it in a program. This is useful when the algorithm is much more compact than the resulting program code. The algorithm in the current example consists of three steps:

1. initialize the variables $v_0$, $g$, and $t$ with numerical values,
2. evaluate $y$ according to the formula (1.1),
3. print the $y$ value to the screen.

The Python program is very close to this text, but some less experienced programmers may want to write the tasks in English before translating them to Python.

The *implementation* of an algorithm is the process of writing and testing a program. The testing phase is also known as *verification*: After the program text is written we need to *verify* that the program works correctly. This is a very important step that will receive substantial attention in the present book. Mathematical software produce numbers, and it is normally quite a challenging task to verify that the numbers are correct.

An *error* in a program is known as a *bug*[13], and the process of locating and removing bugs is called *debugging*. Many look at debugging as the most difficult and challenging part of computer programming.

Programs are built of *statements*. There are many types of statements:

```
v0 = 3
```

is an *assignment* statement, while

```
print y
```

is a *print* statement. It is common to have one statement on each line, but it is possible to write multiple statements on one line if the statements are separated by semi-colon. Here is an example:

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Although most newcomers to computer programming will think they understand the meaning of the lines in the above program, it is important to be aware of some major differences between notation in a

[13] In the very early days of computing, computers were built of a large number of tubes, which glowed and gave off heat. The heat attracted bugs, which caused short circuits. "Debugging" meant shutting down the computer and cleaning out the dead bugs.

computer program and notation in mathematics. When you see the equality sign "=" in mathematics, it has a certain interpretation as an equation ($x+2=5$) or a definition ($f(x)=x^2+1$). In a computer program, however, the equality sign has a quite different meaning, and it is called an *assignment*. The right-hand side of an assignment contains an *expression*, which is a combination of values, variables, and operators. When the expression is *evaluated*, it results in a value that the variable on the left-hand side will refer to. We often say that the right-hand side value is *assigned* to the variable on the left-hand side. In the current context it means that we in the first line assign the number 3 to the variable `v0`, 9.81 to `g`, and 0.6 to `t`. In the next line, the right-hand side expression `v0*t - 0.5*g*t**2` is first evaluated, and the result is then assigned to the `y` variable.

Consider the assignment statement

```
y = y + 3
```

This statement is mathematically false, but in a program it just means that we evaluate the right-hand side expression and assign its value to the variable `y`. That is, we first take the current value of `y` and add 3. The value of this operation is assigned to `y`. The old value of `y` is then lost.

You may think of the `=` as an arrow, `y <- y+3`, rather than an equality sign, to illustrate that the value to the right of the arrow is stored in the variable to the left of the arrow[14]. An example will illustrate the principle of assignment to a variable:

```
y = 3
print y
y = y + 4
print y
y = y*y
print y
```

Running this program results in three numbers: `3`, `7`, `49`. Go through the program and convince yourself that you understand what the result of each statement becomes.

A computer program must have correct *syntax*, meaning that the text in the program must follow the strict rules of the computer language for constructing statements. For example, the syntax of the print statement is the word `print`, followed by one or more spaces, followed by an expression of what we want to print (a Python variable, text enclosed in quotes, a number, for instance). Computers are very picky about syntax! For instance, a human having read all the previous pages may easily understand what this program does,

[14] The R (or S or S-PLUS) programming languages for statistical computing actually use an arrow, while other languages such as Algol, Simula, and Pascal use `:=` to explicitly state that we are not dealing with a mathematical equality.

```
myvar = 5.2
prinnt Myvar
```

but the computer will find two errors in the last line: `prinnt` is an unknown instruction and `Myvar` is an undefined variable. Only the first error is reported (a syntax error), because Python stops the program once an error is found. All errors that Python finds are easy to remove. The difficulty with programming is to remove the rest of the errors, such as errors in formulas or the sequence of operations.

*Blanks* may or may not be important in Python programs. In Chapter 2.1.2 you will see that blanks are in some occasions essential for a correct program. Around `=` or arithmetic operators, however, blanks do not matter. We could hence write our program from Chapter 1.1.7 as

```
v0=3;g=9.81;t=0.6;y=v0*t-0.5*g*t**2;print y
```

This is not a good idea because blanks are essential for easy reading of a program code, and easy reading is essential for finding errors, and finding errors is *the* difficult part of programming. The recommended layout in Python programs specifies one blank around `=`, `+`, and `-`, and no blanks around `*`, `/`, and `**`. Note that the blank after `print` is essential: `print` is a command in Python and `printy` is not recognized as any valid command. (Python would look at `printy` as an undefined variable.) Computer scientists often use the term *whitespace* when referring to a blank[15].

When we interact with computer programs, we usually provide some information to the program and get some information out. It is common to use the term *input data*, or just *input*, for the information that must be known on beforehand. The result from a program is similarly referred to as *output data*, or just *output*. In our example, $v_0$, $g$, and $t$ constitute input, while $y$ is output. All input data must be assigned values in the program before the output can be computed. Input data can be explicitly initialized in the program, as we do in the present example, or the data can be provided by user through keyboard typing while the program is running, as we explain in Chapter 3. Output data can be printed in the terminal window, as in the current example, displayed as graphics on the screen, as done in Chapter 4, or stored in a file for later access, as explained in Chapter 6.

The word *user* usually has a special meaning in computer science: It means a human interacting with a program. You are a user of a text editor for writing Python programs, and you are a user of your

[15] More precisely, blank is the character produced by the space bar on the keyboard, while whitespace denotes any character(s) that, if printed, do not print ink on the paper: a blank, a tabulator character (produced by backslash followed by t), or a newline character (produced by backslash followed by n). The newline character is explained on page 13.

own programs. When you write programs, it is difficult to imagine how other users will interact with the program. Maybe they provide wrong input or misinterpret the output. Making user-friendly programs is very challenging and depends heavily on the target audience of users. The author had the average reader of the book in mind as a typical user when developing programs for this book.

A central part of a computer is the *operating system*. This is actually a collection of programs that manages the hardware and software resources on the computer. There are three major operating systems today: Windows, Macintosh (called Mac for short), and Unix. Several versions of Windows have appeared since the 1990s: Windows 95, 98, 2000, ME, XP, and Vista. Unix was invented already in 1970 and comes in many different versions. Nowadays, two open source implementations of Unix, Linux and Free BSD Unix, are most common. The latter forms the core of the Mac OS X operating system on Macintosh machines, while Linux exists in slightly different flavors: Red Hat, Debian, Ubuntu, and Suse to mention the most important distributions. We will use the term Unix in this book as a synonym for all the operating systems that inherit from classical Unix, such as Solaris, Free BSD, Mac OS X, and any Linux variant. Note that this use of Unix also includes Macintosh machines, but only newer machines as the older ones run an Apple-specific Mac operating system. As a computer user and reader of this book, you should know exactly which operating system you have. In particular, Mac users must know if their operating system is Unix-based or not.

The user's interaction with the operation system is through a set of programs. The most widely used of these enable viewing the contents of folders or starting other programs. To interact with the operating system, as a user, you can either issue commands in a terminal window or use graphical programs. For example, for viewing the file contents of a folder you can run the command `ls` in a Unix terminal window or `dir` in a DOS (Windows) terminal window. The graphical alternatives are many, some of the most common are Windows Explorer on Windows, Nautilus and Konqueror on Unix, and Finder on Mac. To start a program, it is common to double-click on a file icon or write the program's name in a terminal window.

## 1.3 Another Formula: Celsius-Fahrenheit Conversion

Our next example involves the formula for converting temperature measured in Celsius degrees to the corresponding value in Fahrenheit degrees:

$$F = \frac{9}{5}C + 32 \tag{1.2}$$

In this formula, $C$ is the amount of degrees in Celsius, and $F$ is the corresponding temperature measured in Fahrenheit. Our goal now is to write a computer program which can compute $F$ from (1.2) when $C$ is known.

### 1.3.1 Potential Error: Integer Division

*Straightforward Coding of the Formula.* A straightforward attempt at coding the formula (1.2) goes as follows[16]:

```
C = 21
F = (9/5)*C + 32
print F
```

When run, this program prints the value 53. You can find the program in the file `c2f_v1.py`[17] in the `src/formulas` folder – as all other programs from this chapter.

*Verifying the Results.* Testing the correctness is easy in this case since we can evaluate the formula on a calculator: $\frac{9}{5} \cdot 21 + 32$ is 69.8, not 53. What is wrong? The formula in the program looks correct!

*Float and Integer Division.* The error in our program above is one of the most common errors in mathematical software and is not at all obvious for a newcomer to programming. In many computer languages, there are two types of divisions: float division and integer division. Float division is what you know from mathematics: 9/5 becomes 1.8 in decimal notation.

Integer division $a/b$ with integers (whole numbers) $a$ and $b$ results in an integer that is truncated (or mathematically, "rounded down"). More precisely, the result is the largest integer $c$ such that $bc \leq a$. This implies that 9/5 becomes 1 since $1 \cdot 5 = 5 \leq 9$ while $2 \cdot 5 = 10 > 9$. Another example is 1/5, which becomes 0 since $0 \cdot 5 \leq 1$ (and $1 \cdot 5 > 1$). Yet another example is 16/6, which results in 2 (try $2 \cdot 6$ and $3 \cdot 6$ to convince yourself). Many computer languages, including Fortran, C, C++, Java, and Python, interpret a division operation `a/b` as integer division if both operands `a` and `b` are integers. If either `a` or `b` is a real (floating-point) number, `a/b` implies the standard mathematical float division.

The problem with our program is the coding of the formula `(9/5)*C + 32`. This formula is evaluated as follows. First, `9/5` is calculated. Since

[16] The parentheses around `9/5` are not strictly needed, i.e., `(9/5)*C` is computationally identical to `9/5*C`, but parentheses remove any doubt that `9/5*C` could mean `9/(5*C)`. Chapter 1.3.4 has more information on this topic.

[17] The `v1` part of the name stands for "version 1". Throughout this book, we will often develop several trial versions of a program, but remove the version number in the final version of the program.

9 and 5 are interpreted by Python as integers (whole numbers), 9/5 is a division between two integers, and Python chooses by default integer division, which results in 1. Then 1 is multiplied by C, which equals 21, resulting in 21. Finally, 21 and 32 are added with 53 as result.

We shall very soon present a correct version of the temperature conversion program, but first it may be advantageous to introduce a frequently used word in Python programming: *object*.

### 1.3.2 Objects in Python

When we write

```
C = 21
```

Python interprets the number 21 on the right-hand side of the assignment as an integer and creates an int (for integer) *object* holding the value 21. The variable C acts as a *name* for this int object. Similarly, if we write C = 21.0, Python recognizes 21.0 as a real number and therefore creates a float (for floating-point) object holding the value 21.0 and lets C be a name for this object. In fact, any assignment statement has the form of a variable name on the left-hand side and an object on the right-hand side. One may say that Python programming is about solving a problem by defining and changing objects.

At this stage, you do not need to know what an object really is, just think of an int object as a collection, say a storage box, with some information about an integer number. This information is stored somewhere in the computer's memory, and with the name C the program gets access to this information. The fundamental issue right now is that 21 and 21.0 are identical numbers in mathematics, while in a Python program 21 gives rise to an int object and 21.0 to a float object.

There are lots of different object types in Python, and you will later learn how to create your own customized objects. Some objects contain a lot of data, not just an integer or a real number. For example, when we write

```
print 'A text with an integer %d and a float %f' % (2, 2.0)
```

a str (string) object, without a name, is first made of the text between the quotes and then this str object is printed. We can alternatively do this in two steps:

```
s = 'A text with an integer %d and a float %f' % (2, 2.0)
print s
```

### 1.3.3 Avoiding Integer Division

As a quite general rule of thumb, one should avoid integer division in mathematical formulas[18]. There are several ways to do this, as we describe in Appendix E.2. The simplest remedy in Python is to insert a statement that simply turns off integer division. A more widely applicable method, also in other programming languages than Python, is to enforce one of the operands to be a `float` object. In the current example, there are several ways to do this:

```
F = (9.0/5)*C + 32
F = (9/5.0)*C + 32
F = float(C)*9/5 + 32
```

In the first two lines, one of the operands is written as a decimal number, implying a `float` object and hence float division. In the last line, `float(C)*9` means `float` times `int`, which results in a `float` object, and float division is guaranteed.

A related construction,

```
F = float(C)*(9/5) + 32
```

does not work correctly, because `9/5` is evaluated by integer division, yielding 1, before being multiplied by a `float` representation of `C` (see next section for how compound arithmetic operations are calculated). In other words, the formula reads `F=C+32`, which is wrong.

We now understand why the first version of the program does not work and what the remedy is. A correct program is

```
C = 21
F = (9.0/5)*C + 32
print F
```

Instead of `9.0` we may just write `9.` (the dot implies a `float` interpretation of the number). The program is available in the file `c2f.py`. Try to run it – and observe that the output becomes 69.8, which is correct.

*Comment.* We could easily have run into problems in our very first programs if we instead of writing the formula $\frac{1}{2}gt^2$ as `0.5*g*t**2` wrote `(1/2)*g*t**2`. Explain the problem!

### 1.3.4 Arithmetic Operators and Precedence

Formulas in Python programs are usually evaluated in the same way as we would evaluate them mathematically. Python proceeds from left to right, term by term in an expression (terms are separated by plus

[18] Some mathematical algorithms do make use of integer division, but in those cases you should use a double forward slash, `//`, as division operator, because this is Python's way of explicitly indicating integer division.

or minus). In each term, power operations such as $a^b$, coded as `a**b`, has precedence over multiplication and division. As in mathematics, we can use parentheses to dictate the way a formula is evaluated. Below are two illustrations of these principles.

- `5/9+2*a**4/2`: First 5/9 is evaluated (as integer division, giving 0 as result), then $a^4$ (`a**4`) is evaluated, then 2 is multiplied with $a^4$, that result is divided by 2, and the answer is added to the result of the first term. The answer is therefore `a**4`.
- `5/(9+2)*a**(4/2)`: First $\frac{5}{9+2}$ is evaluated (as integer division, yielding 0), then 4/2 is computed (as integer division, yielding 2), then `a**2` is calculated, and that number is multiplied by the result of `5/(9+2)`. The answer is thus always zero.

As evident from these two examples, it is easy to unintentionally get integer division in formulas. Although integer division can be turned off in Python, we think it is important to be strongly aware of the integer division concept and to develop good programming habits to avoid it. The reason is that this concept appears in so many common computer languages that it is better to learn as early as possible how to deal with the problem rather than using a Python-specific feature to remove the problem.

## 1.4 Evaluating Standard Mathematical Functions

Mathematical formulas frequently involve functions such as sin, cos, tan, sinh, cosh, exp, log, etc. On a pocket calculator you have special buttons for such functions. Similarly, in a program you also have ready-made functionality for evaluating these types of mathematical functions. One could in principle write one's own program for evaluating, e.g., the $\sin(x)$ function, but how to do this in an efficient way is a non-trivial topic. Experts have worked on this problem for decades and implemented their best recipes in pieces of software that we should reuse. This section tells you how to reach sin, cos, and similar functions in a Python context.

### 1.4.1 Example: Using the Square Root Function

*Problem.* Consider the vertical motion of a ball in (1.1) on page 1. We now ask the question: How long time does it take for the ball to reach the height $y_c$? The answer is straightforward to derive. When $y = y_c$ we have

$$y_c = v_0 t - \frac{1}{2} g t^2 .$$

We recognize that this equation is a quadratic equation which we must solve with respect to $t$. Rearranging,

$$\frac{1}{2}gt^2 - v_0 t + y_c = 0,$$

and using the well-known formula for the two solutions of a quadratic equation, we find

$$t_1 = \left(v_0 - \sqrt{v_0^2 - 2gy_c}\right)/g, \quad t_2 = \left(v_0 + \sqrt{v_0^2 - 2gy_c}\right)/g\,. \qquad (1.3)$$

There are two solutions because the ball reaches the height $y_c$ on its way up ($t = t_1$) and on its way down ($t = t_2 > t_1$).

*The Program.* To evaluate the expressions for $t_1$ and $t_2$ from (1.3) in a computer program, we need access to the square root function. In Python, the square root function and lots of other mathematical functions, such as sin, cos, sinh, exp, and log, are available in a module called `math`. We must first import the module before we can use it, that is, we must write `import math`. Thereafter, to take the square root of a variable `a`, we can write `math.sqrt(a)`. This is demonstrated in a program for computing $t_1$ and $t_2$:

```
v0 = 5
g = 9.81
yc = 0.2
import math
t1 = (v0 - math.sqrt(v0**2 - 2*g*yc))/g
t2 = (v0 + math.sqrt(v0**2 - 2*g*yc))/g
print 'At t=%g s and %g s, the height is %g m.' % (t1, t2, yc)
```

The output from this program becomes

```
At t=0.0417064 s and 0.977662 s, the height is 0.2 m.
```

You can find the program as the file `ball_yc.py` in the `src/formulas` folder.

*Two Ways of Importing a Module.* The standard way to import a module, say `math`, is to write

```
import math
```

and then access individual functions in the module with the module name as prefix as in

```
x = math.sqrt(y)
```

People working with mathematical functions often find `math.sqrt(y)` less pleasing than just `sqrt(y)`. Fortunately, there is an alternative import syntax that allows us to skip the module name prefix. This alternative syntax has the form “from module import function”. A specific example is

```
from math import sqrt
```

Now we can work with `sqrt` directly, without the `math.` prefix. More than one function can be imported:

```
from math import sqrt, exp, log, sin
```

Sometimes one just writes

```
from math import *
```

to import all functions in the `math` module. This includes `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `exp`, `log` (base $e$), `log10` (base 10), `sqrt`, as well as the famous numbers `e` and `pi`. Importing all functions from a module, using the asterisk (*) syntax, is convenient, but this may result in a lot of extra names in the program that are not used. It is in general recommended not to import more functions than those that are really used in the program[19].

With a `from math import sqrt` statement we can write the formulas for the roots in a more pleasing way:

```
t1 = (v0 - sqrt(v0**2 - 2*g*yc))/g
t2 = (v0 + sqrt(v0**2 - 2*g*yc))/g
```

*Import with New Names.* Imported modules and functions can be given new names in the import statement, e.g.,

```
import math as m
# m is now the name of the math module
v = m.sin(m.pi)

from math import log as ln
v = ln(5)

from math import sin as s, cos as c, log as ln
v = s(x)*c(x) + ln(x)
```

In Python, everything is an object, and variables refer to objects, so new variables may refer to modules and functions as well as numbers and strings. The examples above on new names can also be coded by introducing new variables explicitly:

```
m = math
ln = m.log
s = m.sin
c = m.cos
```

[19] Nevertheless, of convenience we often use the `from module import *` syntax in this book.

### 1.4.2 Example: Using More Mathematical Functions

Our next examples involves calling some more mathematical functions from the `math` module. We look at the definition of the $\sinh(x)$ function:

$$\sinh(x) = \frac{1}{2}\left(e^{x} - e^{-x}\right) . \tag{1.4}$$

We can evaluate $\sinh(x)$ in three ways: i) by calling `math.sinh`, ii) by computing the right-hand side of (1.4), using `math.exp`, or iii) by computing the right-hand side of (1.4) with the aid of the power expressions `math.e**x` and `math.e**(-x)`. A program doing these three alternative calculations is found in the file `3sinh.py`. The core of the program looks like this:

```
from math import sinh, exp, e, pi
x = 2*pi
r1 = sinh(x)
r2 = 0.5*(exp(x) - exp(-x))
r3 = 0.5*(e**x - e**(-x))
print r1, r2, r3
```

The output from the program shows that all three computations give identical results:

```
267.744894041 267.744894041 267.744894041
```

### 1.4.3 A First Glimpse of Round-Off Errors

The previous example computes a function in three different yet mathematically equivalent ways, and the output from the `print` statement shows that the three resulting numbers are equal. Nevertheless, this is not the whole story. Let us try to print out `r1`, `r2`, `r3` with 16 decimals:

```
print '%.16f %.16f %.16f' % (r1,r2,r3)
```

This statement leads to the output

```
267.7448940410164369 267.7448940410164369 267.7448940410163232
```

Now `r1` and `r2` are equal, but `r3` is different! Why is this so?

Our program computes with real numbers, and real numbers need in general an infinite number of decimals to be represented exactly. The computer truncates the sequence of decimals because the storage is finite. In fact, it is quite standard to keep only 16 digits in a real number on a computer. Exactly how this truncation is done is not explained in this book[20]. For now the purpose is to notify the reader that real numbers on a computer often have a small error. Only a few real numbers can be represented exactly with 16 digits, the rest of the real numbers are only approximations.

[20] Instead, you can search for "floating point number" on wikipedia.org.

For this reason, most arithmetic operations involve inaccurate real numbers, resulting in inaccurate calculations. Think of the following two calculations: $1/49 \cdot 49$ and $1/51 \cdot 51$. Both expressions are identical to 1, but when we perform the calculations in Python,

```
print '%.16f %.16f' % (1/49.0*49, 1/51.0*51)
```

the result becomes

```
0.9999999999999999 1.0000000000000000
```

The reason why we do not get exactly 1.0 as answer in the first case, is because 1/49 is not correctly represented in the computer. Also 1/51 has an inexact representation, but the error does not propagate to the final answer.

To summarize, errors[21] in floating-point numbers may propagate through mathematical calculations and result in answers that are only approximations to the exact underlying mathematical values. The errors in the answers are commonly known as *round-off errors*. As soon as you use Python interactively as explained in the next section, you will encounter round-off errors quite often.

Python has a special module `decimal` which allows real numbers to be represented with adjustable accuracy so that round-off errors can be made as small as desired. However, we will hardly use this module[22] because approximations implied by many mathematical methods applied throughout this book normally lead to (much) larger errors than those caused by round-off.

## 1.5 Interactive Computing

A particular convenient feature of Python is the ability to execute statements and evaluate expressions interactively. The environments where you work interactively with programming are commonly known as Python *shells*. The simplest Python shell is invoked by just typing `python` at the command line in a terminal window. Some messages about Python are written out together with a prompt `>>>`, after which you can issue commands. Let us try to use the interactive shell as a calculator. Type in `3*4.5-0.5` and then press the Return key to see Python's response to this expression:

```
Unix/DOS> python
Python 2.5.1 (r251:54863, May  2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 3*4.5-0.5
13.0
```

[21] Exercise 2.49 on page 112 estimates the size of the errors.
[22] See the last paragraph of Chapter 2.2.9 for an example.

The text on a line after >>> is what we write (shell input) and the text without the >>> prompt is the result that Python calculates (shell output). It is easy, as explained below, to recover previous input and edit the text. This editing feature makes it convenient to experiment with statements and expressions.

### 1.5.1 Calculating with Formulas in the Interactive Shell

The program from Chapter 1.1.7 can be typed in, line by line, in the interactive shell:

```
>>> v0 = 5
>>> g = 9.81
>>> t = 0.6
>>> y = v0*t - 0.5*g*t**2
>>> print y
1.2342
```

We can now easily calculate an `y` value corresponding to another (say) `v0` value: hit the up arrow key[23] to recover previous statements, repeat pressing this key until the `v0 = 5` statement is displayed. You can then edit the line, say you edit the statement to

```
>>> v0 = 6
```

Press return to execute this statement. You can control the new value of `v0` by either typing just `v0` or `print v0`:

```
>>> v0
6
>>> print v0
6
```

The next step is to recompute `y` with this new `v0` value. Hit the up arrow key multiple times to recover the statement where `y` is assigned, press the Return key, and write `y` or `print y` to see the result of the computation:

```
>>> y = v0*t - 0.5*g*t**2
>>> y
1.8341999999999996
>>> print y
1.8342
```

The reason why we get two slightly different results is that typing just `y` prints out all the decimals that are stored in the computer (16), while `print y` writes out `y` with fewer decimals. As mentioned on page 25, computations on a computer often suffer from round-off errors. The present calculation is no exception. The correct answer is `1.8342`, but

[23] This key works only if Python was compiled with the Readline library. In case the key does not work, try the editing capabilities in another Python shell, for example, IPython (see Chapter 1.5.3).

round-off errors lead to a number that is incorrect in the 16th decimal. The error is here $4 \cdot 10^{-16}$.

### 1.5.2 Type Conversion

Often you can work with variables in Python without bothering about the type of objects these variables refer to. Nevertheless, we encountered a serious problem in Chapter 1.3.1 with integer division, which forced us to be careful about the types of objects in a calculation. The interactive shell is very useful for exploring types. The following example illustrates the `type` function and how we can convert an object from one type to another.

First, we create an `int` object bound to the name `C` and check its type by calling `type(C)`:

```
>>> C = 21
>>> type(C)
<type 'int'>
```

We convert this `int` object to a corresponding `float` object:

```
>>> C = float(C)   # type conversion
>>> type(C)
<type 'float'>
>>> C
21.0
```

In the statement `C = float(C)` we create a new object from the original object referred to by the name `C` and bind it to the same name `C`. That is, `C` refers to a different object after the statement than before. The original `int` with value `21` cannot be reached anymore (since we have no name for it) and will be automatically deleted by Python.

We may also do the reverse operation, i.e., convert a particular `float` object to a corresponding `int` object:

```
>>> C = 20.9
>>> type(C)
<type 'float'>
>>> D = int(C)      # type conversion
>>> type(D)
<type 'int'>
>>> D
20                  # decimals are truncated :-/
```

In general, one can convert a variable `v` to type `MyType` by writing `v=MyType(v)`, if it makes sense to do the conversion.

In the last input we tried to convert a `float` to an `int`, and this operation implied stripping off the decimals. Correct conversion according to mathematical rounding rules can be achieved with help of the `round` function:

```
>>> round(20.9)
21.0
>>> int(round(20.9))
21
```

### 1.5.3 IPython

There exist several improvements of the standard Python shell presented in the previous section. The author advocates the IPython shell as the preferred interactive shell. You will then need to have IPython installed. Typing `ipython` in a terminal window starts the shell. The (default) prompt in IPython is not `>>>` but `In [X]:`, where `X` is the number of the present input command. The most widely used features of IPython are summarized below.

*Running Programs.* Python programs can be run from within the shell:

```
In [1]: run ball_variables.py
1.2342
```

This command requires that you have taken a `cd` to the folder where the `ball_variables.py` program is located and started IPython from there.

On Windows you may, as an alternative to starting IPython from a DOS window, double click on the IPython desktop icon or use the Start menu. In that case, you must move to the right folder where your program is located. This is done by the `os.chdir` (change directory) command. Typically, you write something like

```
In [1]: import os
In [2]: os.chdir(r'C:\Documents and Settings\me\My Documents\div')
In [3]: run ball_variables.py
```

if the `ball_variables.py` program is located in the folder `div` under `My Documents` of user `me`. Note the `r` before the quote in the string: it is required to let a backslash really mean the backslash character.

We recommend to run all your Python programs from the IPython shell. Especially when something goes wrong, IPython can help you to examine the state of variables so that you become quicker to locate bugs. In the rest of the book, we just write the program name and the output when we illustrate the execution of a program:

Terminal

```
ball_variables.py
1.2342
```

You then need to write `run` before the program name if you execute the program in IPython, or if you prefer to run the program from the

Unix/DOS command prompt in a terminal window, you need to write `python` prior to the program name. Appendix E.1 describes various other ways to run a Python program.

*Quick Recovery of Previous Output.* The results of the previous statements in an interactive IPython session are available in variables of the form `_iX` (underscore, `i`, and a number `X`), where `X` is 1 for the last statement, 2 for the second last statement, and so forth. Short forms are `_` for `_i1`, `__` for `_i2`, and `___` for `_i3`. The output from the `In [1]` input above is `1.2342`. We can now refer to this number by an underscore and, e.g., multiply it by 10:

```
In [2]: _*10
Out[2]: 12.341999999999999
```

Output from Python statements or expressions in IPython are preceded by `Out[X]` where `X` is the command number corresponding to the previous `In [X]` prompt. When programs are executed, as with the `run` command, or when operating system commands are run (as shown below), the output is from the operating system and then not preceded by any `Out[X]` label.

*TAB Completion.* Pressing the TAB key will complete an incompletely typed variable name. For example, after defining `my_long_variable_name = 4`, write just `my` at the `In [4]:` prompt below, and then hit the TAB key. You will experience that `my` is immediately expanded to `my_long_variable_name`. This automatic expansion feature is called TAB completion and can save you from quite some typing.

```
In [3]: my_long_variable_name = 4

In [4]: my_long_variable_name
Out[4]: 4
```

*Recovering Previous Commands.* You can "walk" through the command history by typing `Ctrl-p` or the up arrow for going backward or `Ctrl-n` or the down arrow for going forward. Any command you hit can be edited and re-executed.

*Running Unix/Windows Commands.* Operating system commands can be run from IPython. Below we run the three Unix commands `date`, `ls` (list files), and `mkdir` (make directory):

```
In [5]: date
Thu Nov 18 11:06:16 CET 2010

In [6]: ls
myfile.py  yourprog.py

In [7]: mkdir mytestdir
```

If you have defined Python variables with the same name as operating system commands, e.g., `date=30`, you must write `!date` to run the corresponding operating system command.

IPython can do much more than what is shown here, but the advanced features and the documentation of them probably do not make sense before you are more experienced with Python – and with reading manuals.

*Remark.* In the rest of the book we will apply the `>>>` prompt in interactive sessions instead of the input and output prompts as used by IPython, simply because all Python books and electronic manuals use `>>>` to mark input in interactive shells. However, when you sit by the computer and want to use an interactive shell, we recommend to use IPython, and then you will see the `In [X]` prompt instead of `>>>`.

## 1.6 Complex Numbers

Suppose $x^2 = 2$. Then most of us are able to find out that $x = \sqrt{2}$ is a solution to the equation. The more mathematically interested reader will also remark that $x = -\sqrt{2}$ is another solution. But faced with the equation $x^2 = -2$, very few are able to find a proper solution without any previous knowledge of *complex numbers*. Such numbers have many applications in science, and it is therefore important to be able to use such numbers in our programs.

On the following pages we extend the previous material on computing with real numbers to complex numbers. The text is optional, and readers without knowledge of complex numbers can safely drop this section and jump to Chapter 1.7.

A complex number is a pair of real numbers $a$ and $b$, most often written as $a+bi$, or $a+ib$, where $i$ is called the imaginary unit and acts as a label for the second term. Mathematically, $i = \sqrt{-1}$. An important feature of complex numbers is definitely the ability to compute square roots of negative numbers. For example, $\sqrt{-2} = \sqrt{2}i$ (i.e., $\sqrt{2}\sqrt{-1}$). The solutions of $x^2 = -2$ are thus $x_1 = +\sqrt{2}i$ and $x_2 = -\sqrt{2}i$.

There are rules for addition, subtraction, multiplication, and division between two complex numbers. There are also rules for raising a complex number to a real power, as well as rules for computing $\sin z$, $\cos z$, $\tan z$, $e^z$, $\ln z$, $\sinh z$, $\cosh z$, $\tanh z$, etc. for a complex number $z = a + ib$. We assume in the following that you are familiar with the mathematics of complex numbers, at least to the degree encountered in the program examples.

$$\text{let } u = a + bi \text{ and } v = c + di$$

$$
\begin{aligned}
u = v \quad &: \quad a = c,\ b = d\\
-u &= -a - bi\\
u^* &\equiv a - bi \quad \text{(complex conjugate)}\\
u + v &= (a + c) + (b + d)i\\
u - v &= (a - c) + (b - d)i\\
uv &= (ac - bd) + (bc + ad)i\\
u/v &= \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i\\
|u| &= \sqrt{a^2 + b^2}\\
e^{iq} &= \cos q + i \sin q
\end{aligned}
$$

### 1.6.1 Complex Arithmetics in Python

Python supports computation with complex numbers. The imaginary unit is written as `j` in Python, instead of $i$ as in mathematics. A complex number $2 - 3i$ is therefore expressed as `(2-3j)` in Python. We remark that the number $i$ is written as `1j`, not just `j`. Below is a sample session involving definition of complex numbers and some simple arithmetics:

```
>>> u = 2.5 + 3j       # create a complex number
>>> v = 2              # this is an int
>>> w = u + v          # complex + int
>>> w
(4.5+3j)

>>> a = -2
>>> b = 0.5
>>> s = a + b*1j       # create a complex number from two floats
>>> s = complex(a, b)  # alternative creation
>>> s
(-2+0.5j)
>>> s*w                # complex*complex
(-10.5-3.75j)
>>> s/w                # complex/complex
(-0.25641025641025639+0.28205128205128205j)
```

A `complex` object `s` has functionality for extracting the real and imaginary parts as well as computing the complex conjugate:

```
>>> s.real
-2.0
>>> s.imag
0.5
>>> s.conjugate()
(-2-0.5j)
```

### 1.6.2 Complex Functions in Python

Taking the sine of a complex number does not work:

```
>>> from math import sin
>>> r = sin(w)
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
```

The reason is that the `sin` function from the `math` module only works with real (`float`) arguments, not complex. A similar module, `cmath`, defines functions that take a complex number as argument and return a complex number as result. As an example of using the `cmath` module, we can demonstrate that the relation $\sin(ai) = i \sinh a$ holds:

```
>>> from cmath import sin, sinh
>>> r1 = sin(8j)
>>> r1
1490.4788257895502j
>>> r2 = 1j*sinh(8)
>>> r2
1490.4788257895502j
```

Another relation, $e^{iq} = \cos q + i \sin q$, is exemplified next:

```
>>> q = 8      # some arbitrary number
>>> exp(1j*q)
(-0.14550003380861354+0.98935824662338179j)
>>> cos(q) + 1j*sin(q)
(-0.14550003380861354+0.98935824662338179j)
```

### 1.6.3 Unified Treatment of Complex and Real Functions

The `cmath` functions always return complex numbers. It would be nice to have functions that return a `float` object if the result is a real number and a `complex` object if the result is a complex number. The Numerical Python package (see more about this package in Chapter 4) has such versions of the basic mathematical functions known from `math` and `cmath`. By taking a

```
from numpy.lib.scimath import *
```

one gets access to these flexible versions of mathematical functions[24]. A session will illustrate what we obtain.

Let us first use the `sqrt` function in the `math` module:

```
>>> from math import sqrt
>>> sqrt(4)      # float
2.0
>>> sqrt(-1)     # illegal
Traceback (most recent call last):
  File "<input>", line 1, in ?
ValueError: math domain error
```

[24] The functions also come into play by a `from scipy import *` statement or `from scitools.std import *`. The latter is used as a standard import later in the book.

If we now import sqrt from cmath,

```
>>> from cmath import sqrt
```

the previous sqrt function is overwritten by the new one. More precisely, the name sqrt was previously bound to a function sqrt from the math module, but is now bound to another function sqrt from the cmath module. In this case, any square root results in a complex object:

```
>>> sqrt(4)     # complex
(2+0j)
>>> sqrt(-1)    # complex
1j
```

If we now take

```
>>> from numpy.lib.scimath import *
```

we import (among other things) a new sqrt function. This function is slower than the versions from math and cmath, but it has more flexibility since the returned object is float if that is mathematically possible, otherwise a complex is returned:

```
>>> sqrt(4)     # float
2.0
>>> sqrt(-1)    # complex
1j
```

As a further illustration of the need for flexible treatment of both complex and real numbers, we may code the formulas for the roots of a quadratic function $f(x) = ax^2 + bx + c$:

```
>>> a = 1; b = 2; c = 100   # polynomial coefficients
>>> from numpy.lib.scimath import sqrt
>>> r1 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
>>> r2 = (-b - sqrt(b**2 - 4*a*c))/(2*a)
>>> r1
(-1+9.94987437107j)
>>> r2
(-1-9.94987437107j)
```

Using the up arrow, we may go back to the definitions of the coefficients and change them so the roots become real numbers:

```
>>> a = 1; b = 4; c = 1   # polynomial coefficients
```

Going back to the computations of r1 and r2 and performing them again, we get

```
>>> r1
-0.267949192431
>>> r2
-3.73205080757
```

That is, the two results are float objects. Had we applied sqrt from cmath, r1 and r2 would always be complex objects, while sqrt from the math module would not handle the first (complex) case.

## 1.7 Summary

### 1.7.1 Chapter Topics

*Program Files.* Python programs must be made by a pure text editor such as Emacs, Vim, Notepad++ or similar. The program text must be saved in a text file, with a name ending with the extension `.py`. The filename can be chosen freely, but stay away from names that coincide with modules or keywords in Python, in particular do not use `math.py`, `time.py`, `random.py`, `os.py`, `sys.py`, `while.py`, `for.py`, `if.py`, `class.py`, `def.py`, to mention some forbidden filenames.

*Programs Must Be Accurate!* A program is a collection of statements stored in a text file. Statements can also be executed interactively in a Python shell. Any error in any statement may lead to termination of the execution or wrong results. The computer does exactly what the programmer tells the computer to do!

*Variables.* The statement

```
some_variable = obj
```

defines a variable with the name `some_variable` which refers to an object `obj`. Here `obj` may also represent an expression, say a formula, whose value is a Python object. For example, `1+2.5` involves the addition of an `int` object and a `float` object, resulting in a `float` object. Names of variables can contain upper and lower case English letters, underscores, and the digits from 0 to 9, but the name cannot start with a digit. Nor can a variable name be a reserved word in Python.

If there exists a precise mathematical description of the problem to be solved in a program, one should choose variable names that are in accordance with the mathematical description. Quantities that do not have a defined mathematical symbol, should be referred to by *descriptive* variables names, i.e., names that explain the variable's role in the program. Well-chosen variable names are essential for making a program easy to read, easy to debug, and easy to extend. Well-chosen variable names also reduce the need for comments.

*Comment Lines.* Everything after `#` on a line is ignored by Python and used to insert free running text, known as *comments*. The purpose of comments is to explain, in a human language, the ideas of (several) forthcoming statements so that the program becomes easier to understand for humans. Some variables whose names are not completely self-explanatory also need a comment.

*Object Types.* There are many different types of objects in Python. In this chapter we have worked with

- integers (whole numbers, object type `int`):

```
x10 = 3
XYZ = 2
```

- floats (decimal numbers, object type `float`):

```
max_temperature = 3.0
MinTemp = 1/6.0
```

- strings (pieces of text, object type `str`):

```
a = 'This is a piece of text\nover two lines.'
b = "Strings are enclosed in single or double quotes."
c = """Triple-quoted strings can
span
several lines.
"""
```

- complex numbers (object type `complex`):

```
a = 2.5 + 3j
real = 6; imag = 3.1
b = complex(real, imag)
```

*Operators.* Operators in arithmetic expressions follow the rules from mathematics: power is evaluated before multiplication and division, while the latter two are evaluated before addition and subtraction. These rules are overriden by parentheses. We suggest to use parentheses to group and clarify mathematical expressions, also when not strictly needed.

```
-t**2*g/2
-(t**2)*(g/2)         # equivalent
-t**(2*g)/2           # a different formula!

a = 5.0; b = 5.0; c = 5.0
a/b + c + a*c         # yields 31.0
a/(b + c) + a*c       # yields 25.5
a/(b + c + a)*c       # yields 1.6666666666666665
```

Particular attention must be paid to coding fractions, since the division operator `/` often needs extra parentheses that are not necessary in the mathematical notation for fractions (compare $\frac{a}{b+c}$ with `a/(b+c)` and `a/b+c`).

*Common Mathematical Functions.* The `math` module contains common mathematical functions for real numbers. Modules must be imported before they can be used:

```
import math
a = math.sin(math.pi*1.5)
```

or

```
from math import *
a = sin(pi*1.5)
```

or

```
from math import sin, pi
a = sin(pi*1.5)
```

*Print.* To print the result of calculations in a Python program to a terminal window, we apply the `print` command, i.e., the word `print` followed by a string enclosed in quotes, or just a variable:

```
print "A string enclosed in double quotes"
print a
```

Several objects can be printed in one statement if the objects are separated by commas. A space will then appear between the output of each object:

```
>>> a = 5.0; b = -5.0; c = 1.9856; d = 33
>>> print 'a is', a, 'b is', b, 'c and d are', c, d
a is 5.0 b is -5.0 c and d are 1.9856 33
```

The printf syntax enables full control of the formatting of real numbers and integers:

```
>>> print 'a=%g, b=%12.4E, c=%.2f, d=%5d' % (a, b, c, d)
a=5, b= -5.0000E+00, c=1.99, d=   33
```

Here, `a`, `b`, and `c` are of type `float` and formatted as compactly as possible (`%g` for `a`), in scientific notation with 4 decimals in a field of width 12 (`%12.4E` for `b`), and in decimal notation with two decimals in as compact field as possible (`%.2f` for `c`). The variable `d` is an integer (`int`) written in a field of width 5 characters (`%5d`).

*Integer Division.* A common error in mathematical computations is to divide two integers, because this results in integer division. Any number written without decimals is treated as an integer. To avoid integer division, ensure that every division involves at least one real number, e.g., `9/5` is written as `9.0/5`, `9./5`, `9/5.0`, or `9/5.`.

*Complex Numbers.* Values of complex numbers are written as `(X+Yj)`, where `X` is the value of the real part and `Y` is the value of the imaginary part. One example is `(4-0.2j)`. If the real and imaginary parts are available as variables `r` and `i`, a complex number can be created by `complex(r, i)`.

The `cmath` module must be used instead of `math` if the argument is a complex variable. The `numpy` package offers similar mathematical functions, but with a unified treatment of real and complex variables.

*Terminology.* Some Python and computer science terms briefly covered in this chapter are

- object: anything that a variable (name) can refer to[25] (number, string, function, module, ...)
- variable: name of an object
- statement: an instruction to the computer, usually written on a line in a Python program (multiple statements on a line must be separated by semicolons)
- expression: a combination of numbers, text, variables, and operators that results in a new object, when being evaluated
- assignment: a statement binding an evaluated expression (object) to a variable (name)
- algorithm: detailed recipe for how to solve a problem by programming
- code: program text (or synonym for program)
- implementation: same as code
- executable: the file we run to start the program
- verification: providing evidence that the program works correctly
- debugging: locating and correcting errors in a program

### 1.7.2 Summarizing Example: Trajectory of a Ball

*Problem.* The formula (1.1) computes the height of a ball in vertical motion. What if we throw the ball with an initial velocity having an angle $\theta$ with the horizontal? This problem can be solved by basic high school physics as you are encouraged to do in Exercise 1.14. The ball will follow a trajectory $y = f(x)$ through the air[26], where

$$f(x) = x \tan\theta - \frac{1}{2v_0^2}\frac{gx^2}{\cos^2\theta} + y_0 \,. \tag{1.5}$$

In this expression, $x$ is a horizontal coordinate, $g$ is the acceleration of gravity, $v_0$ is the size of the initial velocity which makes an angle $\theta$ with the $x$ axis, and $(0, y_0)$ is the initial position of the ball. Our programming goal is to make a program for evaluating (1.5). The program should write out the value of all the involved variables and what their units are.

*Solution.* We use the SI system and assume that $v_0$ is given in km/h; $g = 9.81\text{m/s}^2$; $x$, $y$, and $y_0$ are measured in meters; and $\theta$ in degrees.

[25] But objects can exist without being bound to a name: `print 'Hello!'` first makes a string object of the text in quotes and then the contents of this string object, without a name, is printed.

[26] This formula neglects air resistance. Exercise 1.10 explores how important air resistance is. For a soft kick ($v_0 = 10$ km/h) of a football, the gravity force is about 120 times larger than the air resistance. For a hard kick, air resistance may be as important as gravity.

The program has naturally four parts: initialization of input data, import of functions and $\pi$ from `math`, conversion of $v_0$ and $\theta$ to m/s and radians, respectively, and evaluation of the right-hand side expression in (1.5). We choose to write out all numerical values with one decimal. The complete program is found in the file `ball_trajectory.py`:

```
g = 9.81    # m/s**2
v0 = 15     # km/h
theta = 60  # degrees
x = 0.5     # m
y0 = 1      # m

print """\
v0    = %.1f km/h
theta = %d degrees
y0    = %.1f m
x     = %.1f m\
""" % (v0, theta, y0, x)

from math import pi, tan, cos
# convert v0 to m/s and theta to radians:
v0 = v0/3.6
theta = theta*pi/180

y = x*tan(theta) - 1/(2*v0**2)*g*x**2/((cos(theta))**2) + y0

print 'y     = %.1f m' % y
```

The backslash in the triple-quoted multi-line string makes the string continue on the next line without a newline. This means that removing the backslash results in a blank line above the `v0` line and a blank line between the `x` and `y` lines in the output on the screen. Another point to mention is the expression `1/(2*v0**2)`, which might seem as a candidate for unintended integer division. However, the conversion of `v0` to m/s involves a division by 3.6, which results in `v0` being `float`, and therefore `2*v0**2` being `float`. The rest of the program should be self-explanatory at this stage in the book.

We can execute the program in IPython or an ordinary terminal window and watch the output:

Terminal

```
ball_trajectory.py
v0    = 15.0 km/h
theta = 60 degrees
y0    = 1.0 m
x     = 0.5 m
y     = 0.7 m
```

### 1.7.3 About Typesetting Conventions in This Book

This version of the book applies different design elements for different types of "computer text". Complete programs and parts of programs (snippets) are typeset with a light blue background. A snippet looks like this:

```
a = sqrt(4*p + c)
print 'a =', a
```

A complete program has an additional vertical line to the left:

```
C = 21
F = (9.0/5)*C + 32
print F
```

As a reader of this book, you may wonder if a code shown is a complete program you can try out or if it is just a part of a program (a snippet) so that you need to add surrounding statements (e.g., import statements) to try the code out yourself. The appearance of a vertical line to the left or not will then quickly tell you what type of code you see.

An interactive Python session is typeset as

```
>>> from math import *
>>> p = 1; c = -1.5
>>> a = sqrt(4*p + c)
```

Running a program, say `ball_yc.py`, in the terminal window, followed by some possible output is typeset as[27]

Terminal

```
ball_yc.py
At t=0.0417064 s and 0.977662 s, the height is 0.2 m.
```

Sometimes just the output from a program is shown, and this output appears as plain "computer text":

```
h = 0.2
order=0, error=0.221403
order=1, error=0.0214028
order=2, error=0.00140276
order=3, error=6.94248e-05
order=4, error=2.75816e-06
```

Files containing data are shown in a similar way in this book:

```
date   Oslo   London   Berlin   Paris   Rome   Helsinki
01.05  18     21.2     20.2     13.7    15.8   15
01.06  21     13.2     14.9     18      24     20
01.07  13     14       16       25      26.2   14.5
```

## 1.8 Exercises

### What Does It Mean to Solve an Exercise?

The solution to most of the exercises in this book is a Python program. To produce the solution, you first need understand the problem and

[27] Recall from Chapter 1.5.3 that we just write the program name. A real execution demands prefixing the program name by `python` in a DOS/Unix terminal window, or by `run` if you run the program from an interactive IPython session. We refer to Appendix E.1 for more complete information on running Python programs in different ways.

what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements. Equally important is the verification (testing) of the program. A complete solution to a programming exercises therefore consists of two parts: the program text and a demonstration that the program works correctly. Some simple programs, like the ones in the first two exercises below, have so simple output that the verification can just be to run the program and record the output.

In cases where the correctness of the output is not obvious, it is necessary to provide information together with the output to “prove” that the result is correct. This can be a calculation done separately on a calculator, or one can apply the program to a special simple test with known results. The requirement is to provide evidence that the program works as intended.

The sample run of the program to check its correctness can be inserted at the end of the program as a triple-quoted string[28]. The contents of the string can be text from the run in the terminal window, cut and pasted to the program file by the aid of the mouse. Alternatively, one can run the program and direct the output to a file[29]:

Terminal

```
Unix/DOS> python myprogram.py > result
```

Afterwards, use the editor to insert the file `result` inside the string.

As an example, suppose we are to write a program for converting Fahrenheit degrees to Celsius. The solution process can be divided into three steps:

1. Establish the mathematics to be implemented: solving (1.2) with respect to $C$ gives the conversion formula

$$C = \frac{5}{9}(F - 32)\,.$$

2. Coding of the formula in Python: `C = (5.0/9)*(F - 32)`
3. Establish a test case: from the `c2f.py` program in Chapter 1.3.3 we know that $C = 21$ corresponds to $F = 69.8$. We can therefore, in our new program, set $F = 69.8$ and check that $C = 21$. The output from a run can be appended as a triple quoted string at the end of the program.

[28] Alternatively, the output lines can be inserted as comments, but using a multi-line string requires less typing. (Technically, a string object is created, but not assigned to any name or used for anything in the program – but for a human the text in the string contains useful information.)

[29] The redirection to files does not work if the program is run inside IPython. In a DOS terminal window you may also choose to redirect output to a file, because cut and paste between the DOS window and the program window does not work by default unless you right-click the top bar, choose Properties and tick off Quick Edit Mode.

An appropriate complete solution to the exercise is then

```
# Convert from Fahrenheit degrees to Celsius degrees:
F = 69.8
C = (5.0/9)*(F - 32)
print C

'''
Sample run:
python f2c.py
21.0
'''
```

Another way of documenting the output from your own program is to use the `pyreport` program, which formats the code nicely and inserts the result of all output from the program in the resulting report. Applying `pyreport` to the `f2c.py` program is very simple:

Terminal
```
Unix/DOS> pyreport f2c.py
```

The result is a file `f2c.pdf` which you can print. Figure 1.5 displays what the printed file looks like. You can also generate a web page instead of a PDF file[30]:

Terminal
```
Unix/DOS> pyreport -t html f2c.py
```

The result now is a file `f2c.html` which you can load into a web browser. The `pyreport` program can do much more than shown in Figure 1.5. For example, mathematical formulas and graphics can easily be inserted in the resulting document[31].

/home/some/user/intro-programming/work/f2c.py 1

```
F = 69.8
C = (5.0/9)*(F - 32)
print C
```

```
21.0
```

**Fig. 1.5** Output from `pyreport`.

**Exercise 1.1.** *Compute 1+1.*

The first exercise concerns some basic mathematics: Write a Python program for printing the result of 1+1. Name of program file: `1plus1.py`. ◇

[30] The `-t` option specifies the output file type, which here is `html` – the common language in web pages. By default, the output from `pyreport` is PDF. Many other formats and options are possible, just write `pyreport` to see the possibilities.

[31] You can search for "pyreport" on google – the first hit leads you to a description of the program.

**Exercise 1.2.** *Write a "Hello, World!" program.*

Almost all books about programming languages start with a very simple program that prints the text "Hello, World!" to the screen. Make such a program in Python. Name of program file: `hello_world.py`. ◇

**Exercise 1.3.** *Convert from meters to British length units.*

Make a program where you set a length given in meters and then compute and write out the corresponding length measured in inches, in feet, in yards, and in miles. Use that one inch is 2.54 cm, one foot is 12 inches, one yard is 3 feet, and one British mile is 1760 yards. As a verification, a length of 640 meters corresponds to 25196.85 inches, 2099.74 feet, 699.91 yards, or 0.3977 miles. Name of program file: `length_conversion.py`. ◇

**Exercise 1.4.** *Compute the mass of various substances.*

The density of a substance is defined as $\varrho = m/V$, where $m$ is the mass of a volume $V$. Compute and print out the mass of one liter of each of the following substances whose densities in $\mathrm{g/cm}^3$ are found in the file `src/files/densities.dat`: iron, air, gasoline, ice, the human body, silver, and platinum: 21.4. Name of program file: `1liter.py`. ◇

**Exercise 1.5.** *Compute the growth of money in a bank.*

Let $p$ be a bank's interest rate in percent per year. An initial amount $A$ has then grown to

$$A\left(1+\frac{p}{100}\right)^n$$

after $n$ years. Make a program for computing how much money 1000 euros have grown to after three years with 5% interest rate. Name of program file: `interest_rate.py`. ◇

**Exercise 1.6.** *Find error(s) in a program.*

Suppose somebody has written a simple one-line program for computing sin(1):

```
x=1; print 'sin(%g)=%g' % (x, sin(x))
```

Type in this program and try to run it. What is the problem? ◇

**Exercise 1.7.** *Type in program text.*

Type the following program in your editor and execute it. If your program does not work, check that you have copied the code correctly.

```
from math import pi

h = 5.0   # height
b = 2.0   # base
r = 1.5   # radius

area_parallelogram = h*b
print 'The area of the parallelogram is %.3f' % area_parallelogram
```

```
area_square = b**2
print 'The area of the square is %g' % area_square

area_circle = pi*r**2
print 'The area of the circle is %8.3f' % area_circle

volume_cone = 1.0/3*pi*r**2*h
print 'The volume of the cone is %.3f' % volume_cone
```

Name of program file: `formulas_shapes.py`. ◇

**Exercise 1.8.** *Type in programs and debug them.*

Type these short programs in your editor and execute them. When they do not work, identify and correct the erroneous statements.

(a)Does $\sin^2(x) + \cos^2(x) = 1$?

```
from math import sin, cos
x = pi/4
1_val = sin^2(x) + cos^2(x)
print 1_VAL
```

Name of program file: `sin2_plus_cos2.py`

(b)Work with the expressions for movement with constant acceleration:

```
v0 = 3 m/s
t = 1 s
a = 2 m/s**2
s = v0*t + 1/2 a*t**2
print s
```

Name of program file: `acceleration.py`

(c)Verify these equations:

$$(a + b)^2 = a^2 + 2ab + b^2$$

$$(a - b)^2 = a^2 - 2ab + b^2$$

```
a = 3,3   b = 5,3
a2 = a**2
b2 = b**2

eq1_sum = a2 + 2ab + b2
eq2_sum = a2 - 2ab + b2

eq1_pow = (a + b)**2
eq2_pow = (a - b)**2

print 'First equation:  %g = %g', % (eq1_sum, eq1_pow)
print 'Second equation: %h = %h', % (eq2_pow, eq2_pow)
```

Name of program file: `a_pm_b_sqr.py` ◇

**Exercise 1.9.** *Evaluate a Gaussian function.*

The bell-shaped Gaussian function,

$$f(x) = \frac{1}{\sqrt{2\pi}\, s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right], \tag{1.6}$$

is one of the most widely used functions in science and technology[32]. The parameters $m$ and $s$ are real numbers, where $s$ must be greater than zero. Make a program for evaluating this function when $m = 0$, $s = 2$, and $x = 1$. Verify the program's result by comparing with hand calculations on a calculator. Name of program file: `Gaussian_function1.py`. ◇

**Exercise 1.10.** *Compute the air resistance on a football.*

The drag force, due to air resistance, on an object can be expressed as

$$F_d = \frac{1}{2} C_D \varrho A V^2, \tag{1.7}$$

where $\varrho$ is the density of the air, $V$ is the velocity of the object, $A$ is the cross-sectional area (normal to the velocity direction), and $C_D$ is the drag coefficient, which depends heavily on the shape of the object and the roughness of the surface.

The gravity force on an object with mass $m$ is $F_g = mg$, where $g = 9.81\text{m}\,\text{s}^{-2}$.

Make a program that computes the drag force and the gravity force on an object. Write out the forces with one decimal in units of Newton ($\text{N} = \text{kg}\,\text{m/s}^2$). Also print the ratio of the drag force and the gravity force. Define $C_D$, $\varrho$, $A$, $V$, $m$, $g$, $F_d$, and $F_g$ as variables, and put a comment with the corresponding unit.

As a computational example, you can initialize all variables with values relevant for a football kick. The density of air is $\varrho = 1.2$ kg $\text{m}^{-3}$. For any ball, we have obviously that $A = \pi a^2$, where $a$ is the radius of the ball, which can be taken as 11 cm for a football. The mass of the ball is 0.43 kg. $C_D$ can be taken as 0.2.

Use the program to calculate the forces on the ball for a hard kick, $V = 120$ km/h and for a soft kick, $V = 10$ km/h (it is easy to mix inconsistent units, so make sure you compute with $V$ expressed in m/s). Name of program file: `kick.py`. ◇

**Exercise 1.11.** *Define objects in IPython.*

Start `ipython` and give the following command, which will store the interactive session to a file `mysession.log`:

[32] The function is named after Carl Friedrich Gauss, 1777–1855, who was a German mathematician and scientist, now considered as one of the greatest scientists of all time. He contributed to many fields, including number theory, statistics, mathematical analysis, differential geometry, geodesy, electrostatics, astronomy, and optics. Gauss introduced the function (1.6) when he analyzed probabilities related to astronomical data.

```
In [1]: %logstart -r -o mysession.log
```

Thereafter, define an integer, a real number, and a string in IPython. Apply the `type` function to check that each object has the right type. Print the three objects using printf syntax. Finally, type `logoff` to end the recording of the interactive session:

```
In [8]: %logoff
```

Leave IPython and restart it with the `-logplay mysession.log` on the command line. IPython will now re-execute the input statements in the logfile `mysession.log` so that you get back the variables you declared. Print out the variables to demonstrate this fact. ◇

**Exercise 1.12.** *How to cook the perfect egg.*

As an egg cooks, the proteins first denature and then coagulate. When the temperature exceeds a critical point, reactions begin and proceed faster as the temperature increases. In the egg white the proteins start to coagulate for temperatures above 63 C, while in the yolk the proteins start to coagulate for temperatures above 70 C. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 C, but the yolk should not be heated above 70 C. For a hard boiled egg, the center of the yolk should be allowed to reach 70 C.

The following formula expresses the time $t$ it takes (in seconds) for the center of the yolk to reach the temperature $T_y$ (in Celsius degrees):

$$t = \frac{M^{2/3}c\rho^{1/3}}{K\pi^2(4\pi/3)^{2/3}} \ln\left[0.76\frac{T_o - T_w}{T_y - T_w}\right] . \tag{1.8}$$

Here, $M$, $\rho$, $c$, and $K$ are properties of the egg: $M$ is the mass, $\rho$ is the density, $c$ is the specific heat capacity, and $K$ is thermal conductivity. Relevant values are $M = 47$ g for a small egg and $M = 67$ g for a large egg, $\rho = 1.038\ \mathrm{g\,cm^{-3}}$, $c = 3.7\ \mathrm{J\,g^{-1}\,K^{-1}}$, and $K = 5.4\cdot 10^{-3}\ \mathrm{W\,cm^{-1}\,K^{-1}}$. Furthermore, $T_w$ is the temperature (in C degrees) of the boiling water, and $T_o$ is the original temperature (in C degrees) of the egg before being put in the water. Implement the formula in a program, set $T_w = 100$ C and $T_y = 70$ C, and compute $t$ for a large egg taken from the fridge ($T_o = 4$ C) and from room temperature ($T_o = 20$ C). Name of program file: `egg.py`. ◇

**Exercise 1.13.** *Evaluate a function defined by a sum.*

The piecewise constant function

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t < T \end{cases} \tag{1.9}$$

can be approximated by the sum

$$S(t;n) = \frac{4}{\pi}\sum_{i=1}^{n}\frac{1}{2i-1}\sin\frac{2(2i-1)\pi t}{T} \tag{1.10}$$

$$= \frac{4}{\pi}\left(\sin\frac{2\pi t}{T} + \frac{1}{3}\sin\frac{6\pi t}{T} + \frac{1}{5}\sin\frac{10\pi t}{T} + \cdots\right) \tag{1.11}$$

It can be shown that $S(t;n) \rightarrow f(t)$ as $n \rightarrow \infty$. Write a program that prints out the value of $S(\alpha T;n)$ for $\alpha = 0.01$, $T = 2\pi$, and $n = 1,2,3,4$. Let `S` ($= S(t;n)$), `t`, `alpha`, and `T` be variables in the program. A new `S`, corresponding to a new $n$, should be computed by adding one term to the previous value of `S`, i.e., by a statement like `S = S + term`. Run the program also for $\alpha = 1/4$. Does the approximation $S(\alpha T;4)$ seem to be better for $\alpha = 1/4$ than for $\alpha = 0.01$? Name of program file: `compare_func_sum.py`.

*Remark.* This program is very tedious to write for large $n$, but with a loop, introduced in the next chapter (see Exercise 2.39), we can just code the generic term parameterized by $i$ in (1.10). This approach yields a short program that can be used to evaluate $S(t;n)$ for any $n$. Exercise 4.20 extends such an implementation with graphics for displaying how well $S(t;n)$ approximates $f(t)$ for some values of $n$. A sum of sine and/or cosine functions, as in (1.10), is called a *Fourier series*. Approximating a function by a Fourier series is a very important technique in science and technology. ◇

**Exercise 1.14.** *Derive the trajectory of a ball.*

The purpose of this exercise is to explain how Equation (1.5) for the trajectory of a ball arises from basic physics. There is no programming in this exercise, just physics and mathematics.

The motion of the ball is governed by Newton's second law:

$$F_x = ma_x \tag{1.12}$$

$$F_y = ma_y \tag{1.13}$$

where $F_x$ and $F_y$ are the sum of forces in the $x$ and $y$ directions, respectively, $a_x$ and $a_y$ are the accelerations of the ball in the $x$ and $y$ directions, and $m$ is the mass of the ball. Let $(x(t), y(t))$ be the position of the ball, i.e., the horisontal and vertical corrdinate of the ball at time $t$. There are well-known relations between acceleration, velocity, and position: the acceleration is the time derivative of the velocity, and the velocity is the time derivative of the position. Therefore we have that

$$a_x = \frac{d^2x}{dt^2}, \tag{1.14}$$

$$a_y = \frac{d^2y}{dt^2}. \tag{1.15}$$

If we assume that gravity is the only important force on the ball, $F_x = 0$ and $F_y = -mg$.

Integrate the two components of Newton's second law twice. Use the initial conditions on velocity and position,

$$\frac{d}{dt}x(0) = v_0 \cos\theta, \tag{1.16}$$

$$\frac{d}{dt}y(0) = v_0 \sin\theta, \tag{1.17}$$

$$x(0) = 0, \tag{1.18}$$

$$y(0) = y_0, \tag{1.19}$$

to determine the four integration constants. Write up the final expressions for $x(t)$ and $y(t)$. Show that if $\theta = \pi/2$, i.e., the motion is purely vertical, we get the formula (1.1) for the $y$ position. Also show that if we eliminate $t$, we end up with the relation (1.5) between the $x$ and $y$ coordinates of the ball. You may read more about this type of motion in a physics book, e.g., [6]. ◇

**Exercise 1.15.** *Find errors in the coding of formulas.*

Some versions of our program for calculating the formula (1.2) are listed below. Determine which versions that will not work correctly and explain why in each case.

```
C = 21;     F =  9/5*C + 32;        print F
C = 21.0;   F =  (9/5)*C + 32;      print F
C = 21.0;   F =  9*C/5 + 32;        print F
C = 21.0;   F =  9.*(C/5.0) + 32;   print F
C = 21.0;   F =  9.0*C/5.0 + 32;    print F
C = 21;     F =  9*C/5 + 32;        print F
C = 21.0;   F =  (1/5)*9*C + 32;    print F
C = 21;     F =  (1./5)*9*C + 32;   print F
```

◇

**Exercise 1.16.** *Find errors in Python statements.*

Try the following statements in an interactive Python shell. Explain why some statements fail and correct the errors.

```
1a = 2
a1 = b
x = 2
y = X + 4  # is it 6?
from Math import tan
print tan(pi)
pi = "3.14159'
print tan(pi)
c = 4**3**2**3
_ = ((c-78564)/c + 32))
discount = 12%
AMOUNT = 120.-
amount = 120$
```

```
address = hpl@simula.no
and = duck
class = 'INF1100, gr 2"
continue_ = x > 0
bærtype = """jordbær"""
rev = fox = True
Norwegian = ['a human language']
true = fox is rev in Norwegian
```

Hint: It might be wise to test the values of the expressions on the right-hand side, and the validity of the variable names, seperately before you put the left- and right-hand sides together in statements. The last two statements work, but explaining why goes beyond what is treated in this chapter. ◇

**Exercise 1.17.** *Find errors in the coding of a formula.*

Given a quadratic equation,

$$ax^2 + bx + c = 0,$$

the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,. \tag{1.20}$$

Why does the following program not work correctly?

```
a = 2; b = 1; c = 2
from math import sqrt
q = sqrt(b*b - 4*a*c)
x1 = (-b + q)/2*a
x2 = (-b - q)/2*a
print x1, x2
```

Hint: Compute all terms in (1.20) with the aid of a calculator, and compare with the corresponding intermediate results computed in the program (you need to add some `print` statements to see the result of `q`, `-b+q`, and `2*a`). ◇

# 2 Basic Constructions

This chapter introduces some fundamental topics in programming: list objects, `while` and `for` loops, `if-else` branches, and user-defined functions. Everything covered here will be essential for programming in general - and of course in the rest of the book. The programs associated with the chapter are found in the folder `src/basic`.

## 2.1 Loops and Lists for Tabular Data

The goal of our next programming example is to print out a conversion table with Celsius degrees in the first column of the table and the corresponding Fahrenheit degrees in the second column:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  5  41.0
 10  50.0
 15  59.0
 20  68.0
 25  77.0
 30  86.0
 35  95.0
 40 104.0
```

### 2.1.1 A Naive Solution

Since we know how to evaluate the formula (1.2) for one value of $C$, we can just repeat these statements as many times as required for the table above. Using three statements per line in the program, for compact layout of the code, we can write the whole program as

```
C = -20;  F = 9.0/5*C + 32;  print C, F
C = -15;  F = 9.0/5*C + 32;  print C, F
C = -10;  F = 9.0/5*C + 32;  print C, F
C =  -5;  F = 9.0/5*C + 32;  print C, F
C =   0;  F = 9.0/5*C + 32;  print C, F
C =   5;  F = 9.0/5*C + 32;  print C, F
C =  10;  F = 9.0/5*C + 32;  print C, F
C =  15;  F = 9.0/5*C + 32;  print C, F
C =  20;  F = 9.0/5*C + 32;  print C, F
C =  25;  F = 9.0/5*C + 32;  print C, F
C =  30;  F = 9.0/5*C + 32;  print C, F
C =  35;  F = 9.0/5*C + 32;  print C, F
C =  40;  F = 9.0/5*C + 32;  print C, F
```

Running this program, which is stored in the file `c2f_table_repeat.py`, demonstrates that the output becomes

```
-20 -4.0
-15 5.0
-10 14.0
-5 23.0
0 32.0
5 41.0
10 50.0
15 59.0
20 68.0
25 77.0
30 86.0
35 95.0
40 104.0
```

This output suffers from somewhat ugly formatting, but that problem can quickly be fixed by replacing `print C, F` by a `print` statement based on printf formatting. We will return to this detail later.

The main problem with the program above is that lots of statements are identical and repeated. First of all it is boring to write this sort of repeated statements, especially if we want many more $C$ and $F$ values in the table. Second, the idea of the computer is to automate repetition. Therefore, all computer languages have constructs to efficiently express repetition. These constructs are called *loops* and come in two variants in Python: `while` loops and `for` loops. Most programs in this book employ loops, so this concept is extremely important to learn.

### 2.1.2 While Loops

The `while` loop is used to repeat a set of statements as long as a condition is true. We shall introduce this kind of loop through an example. The task is to generate the rows of the table of $C$ and $F$ values. The $C$ value starts at $-20$ and is incremented by 5 as long as $C \leq 40$. For each $C$ value we compute the corresponding $F$ value and write out the two temperatures. In addition, we also add a line of hyphens above and below the table. We postpone to nicely format the $C$ and $F$ columns of numbers and perform for simplicity a plain `print C, F` statement inside the loop.

Using a mathematical type of notation, we could write the `while` loop as follows:

$C = -20$
while $C \leq 40$ repeat the following:
    $F = \frac{9}{5}C + 32$
    print $C$, $F$
    set $C$ to $C + 5$

The three lines after the "while" line are to be repeated as long as the condition $C \leq 40$ is true. This algorithm will then produce a table of $C$ and corresponding $F$ values.

A complete Python program, implementing the repetition algorithm above, looks quite similar[1]:

```
print '------------------'      # table heading
C = -20                         # start value for C
dC = 5                          # increment of C in loop
while C <= 40:                  # loop heading with condition
    F = (9.0/5)*C + 32          # 1st statement inside loop
    print C, F                  # 2nd statement inside loop
    C = C + dC                  # 3rd statement inside loop
print '------------------'      # end of table line (after loop)
```

A very important feature of Python is now encountered: The block of statements to be executed in each pass of the `while` loop must be indented. In the example above the block consists of three lines, and all these lines must have exactly the same indentation. Our choice of indentation in this book is four spaces. The first statement whose indentation coincides with that of the `while` line marks the end of the loop and is executed after the loop has terminated. In this example this is the final `print` statement. You are encouraged to type in the code above in a file, indent the last line four spaces, and observe what happens (you will experience that lines in the table are separated by a line of dashes: `-----`).

Many novice Python programmers forget the colon at the end of the `while` line – this colon is essential and marks the beginning of the indented block of statements inside the loop. Later, we will see that there are many other similar program constructions in Python where there is a heading ending with a colon, followed by an indented block of statements.

Programmers need to fully understand what is going on in a program and be able to simulate the program by hand. Let us do this with the program segment above. First, we define the start value for the sequence of Celsius temperatures: `C = -20`. We also define the increment `dC` that will be added to `C` inside the loop. Then we enter the loop condition `C <= 40`. The first time `C` is `-20`, which implies that `C <= 40` (equivalent to $C \leq 40$ in mathematical notation) is true. Since the loop condition is true, we enter the loop and execute all the indented state-

[1] For this table we also add (of teaching purposes) a line above and below the table.

ments. That is, we compute `F` corresponding to the current `C` value, print the temperatures, and increment `C` by `dC`.

Thereafter, we enter the second pass in the loop. First we check the condition: `C` is `-15` and `C <= 40` is still true. We execute the statements in the indented loop block, `C` becomes `-10`, this is still less than or equal to `40`, so we enter the loop block again. This procedure is repeated until `C` is updated from `40` to `45` in the final statement in the loop block. When we then test the condition, `C <= 40`, this condition is no longer true, and the loop is terminated. We proceed with the next statement that has the same indentation as the `while` statement, which is the final `print` statement in this example.

Newcomers to programming are sometimes confused by statements like

```
C = C + dC
```

This line looks erroneous from a mathematical viewpoint, but the statement is perfectly valid computer code, because we first evaluate the expression on the right-hand side of the equality sign and then let the variable on the left-hand side refer to the result of this evaluation. In our case, `C` and `dC` are two different `int` objects. The operation `C+dC` results in a new `int` object, which in the assignment `C = C+dC` is bound to the name `C`. Before this assignment, `C` was already bound to a `int` object, and this object is automatically destroyed when `C` is bound to a new object and there are no other names (variables) referring to this previous object[2].

Since incrementing the value of a variable is frequently done in computer programs, there is a special short-hand notation for this and related operations:

```
C += dC   # equivalent to C = C + dC
C -= dC   # equivalent to C = C - dC
C *= dC   # equivalent to C = C*dC
C /= dC   # equivalent to C = C/dC
```

### 2.1.3 Boolean Expressions

In our example regarding a `while` loop we worked with a condition `C <= 40`, which evaluates to either true or false, written as `True` or `False` in Python. Other comparisons are also useful:

```
C == 40    # C equals 40
C != 40    # C does not equal 40
C >= 40    # C is greater than or equal to 40
C >  40    # C is greater than 40
C <  40    # C is less than 40
```

[2] If you did not get the last point here, just relax and continue reading.

Not only comparisons between numbers can be used as conditions in `while` loops: Any expression that has a boolean (`True` or `False`) value can be used. Such expressions are known as *logical* or *boolean* expressions.

The keyword `not` can be inserted in front of the boolean expression to change the value from `True` to `False` or from `False` to `True`. To evaluate `not C == 40`, we first evaluate `C == 40`, say this is `True`, and then `not` turns the value into `False`. On the opposite, if `C == 40` is `False`, `not C == 40` becomes `True`. Mathematically it is easier to read `C != 40` than `not C == 40`, but these two boolean expressions are equivalent.

Boolean expressions can be combined with `and` and `or` to form new compound boolean expressions, as in

```
while x > 0 and y <= 1:
    print x, y
```

If `cond1` and `cond2` are two boolean expressions with values `True` or `False`, the compound boolean expression `cond1 and cond2` is `True` if both `cond1` and `cond2` are `True`. On the other hand, `cond1 or cond2` is `True` if at least one of the conditions, `cond1` or `cond2`, is `True`[3]

Here are some more examples from an interactive session where we just evaluate the boolean expressions themselves without using them in loop conditions:

```
>>> x = 0;  y = 1.2
>>> x >= 0 and y < 1
False
>>> x >= 0 or y < 1
True
>>> x > 0 or y > 1
True
>>> x > 0 or not y > 1
False
>>> -1 < x <= 0   #  -1 < x and x <= 0
True
>>> not (x > 0 or y > 0)
False
```

In the last sample expression, `not` applies to the value of the boolean expression inside the parentheses: `x>0` is `False`, `y>0` is `True`, so the combined expression with `or` is `True`, and `not` turns this value to `False`.

The common[4] boolean values in Python are `True`, `False`, `0` (false), and any integer different from zero (true). To see such values in action, we recommend to do Exercises 2.54 and 2.47.

[3] In Python, `cond1 and cond2` or `cond1 or cond2` returns one of the operands and not just `True` or `False` values as in most other computer languages. The operands `cond1` or `cond2` can be expressions or objects. In case of expressions, these are first evaluated to an object before the compound boolean expression is evaluated. For example, `(5+1) or -1` evaluates to `6` (the second operand is not evaluated when the first one is `True`), and `(5+1) and -1` evaluates to `-1`.

[4] All objects in Python can in fact be evaluated in a boolean context, and all are `True` except `False`, zero numbers, and empty strings, lists, and dictionaries. See Exercise 6.27 for more details.

Erroneous thinking about boolean expressions is one of the most common sources of errors in computer programs, so you should be careful every time you encounter a boolean expression and check that it is correctly stated.

### 2.1.4 Lists

Up to now a variable has typically contained a single number. Sometimes numbers are naturally grouped together. For example, all Celsius degrees in the first column of our table could be conveniently stored together as a group. A Python *list* can be used to represent such a group of numbers in a program. With a variable that refers to the list, we can work with the whole group at once, but we can also access individual elements of the group. Figure 2.1 illustrates the difference between an `int` object and a list object. In general, a list may contain a sequence of arbitrary objects. Python has great functionality for examining and manipulating such sequences of objects, which will be demonstrated below.

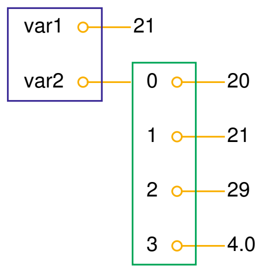



**Fig. 2.1** Illustration of two variables: `var1` refers to an `int` object with value 21, created by the statement `var1 = 21`, and `var2` refers to a `list` object with value `[20, 21, 29, 4.0]`, i.e., three `int` objects and one `float` object, created by the statement `var2 = [20, 21, 29, 4.0]`.

To create a list with the numbers from the first column in our table, we just put all the numbers inside square brackets and separate the numbers by commas:

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

The variable `C` now refers to a `list` object holding 13 list *elements*. All list elements are in this case `int` objects.

Every element in a list is associated with an *index*, which reflects the position of the element in the list. The first element has index 0, the second index 1, and so on. Associated with the `C` list above we have 13 indices, starting with 0 and ending with 12. To access the element with index 3, i.e., the fourth element in the list, we can write `C[3]`. As we see from the list, `C[3]` refers to an `int` object with the value $-5$.

Elements in lists can be deleted, and new elements can be inserted anywhere. The functionality for doing this is built into the list object and accessed by a dot notation. Two examples are `C.append(v)`, which appends a new element `v` to the end of the list, and `C.insert(i,v)`, which inserts a new element `v` in position number `i` in the list. The number of elements in a list is given by `len(C)`. Let us exemplify some list operations in an interactive session to see the effect of the operations:

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]     # create list
>>> C.append(35)               # add new element 35 at the end
>>> C                          # view list C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
```

Two lists can be added:

```
>>> C = C + [40, 45]           # extend C at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

What adding two lists means is up to the list object to define[5], but not surprisingly, addition of two lists is defined as appending the second list to the first. The result of `C + [40,45]` is a new list object, which we then assign to `C` such that this name refers to this new list.

New elements can in fact be inserted anywhere in the list (not only at the end as we did with `C.append`):

```
>>> C.insert(0, -15)           # insert new element -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

With `del C[i]` we can remove an element with index `i` from the list `C`. Observe that this changes the list, so `C[i]` refers to another (the next) element after the removal:

```
>>> del C[2]                   # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]                   # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)                     # length of list
11
```

The command `C.index(10)` returns the index corresponding to the first element with value `10` (this is the 4th element in our sample list, with index 3):

[5] Every object in Python and everything you can do with them is defined by programs made by humans. With the techniques of Chapter 7 you can create your own objects and define (if desired) what it means to add such objects. All this gives enormous power in the hands of programmers. As one example, you can easily define your own list objects if you are not satisfied with Python's own lists.

```
>>> C.index(10)                # find index for an element (10)
3
```

To just test if an object with the value `10` is an element in the list, one can write the boolean expression `10 in C`:

```
>>> 10 in C                    # is 10 an element in C?
True
```

Python allows negative indices, which "count from the right". As demonstrated below, `C[-1]` gives the last element of the list `C`. `C[-2]` is the element before `C[-1]`, and so forth.

```
>>> C[-1]                      # view the last list element
45
>>> C[-2]                      # view the next last list element
40
```

There is a compact syntax for creating variables that refer to the various list elements. Simply list a sequence of variables on the left-hand side of an assignment to a list:

```
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

The number of variables on the left-hand side must match the number of elements in the list, otherwise an error occurs.

A final comment regards the syntax: some list operations are reached by a dot notation, as in `C.append(e)`, while other operations requires the list object as an argument to a function, as in `len(C)`. Although `C.append` for a programmer behaves as a function, it is a function that is reached through a list object, and it is common to say that `append` is a *method* in the list object, not a function. There are no strict rules in Python whether functionality regarding an object is reached through a method or a function.

### 2.1.5 For Loops

*The Nature of For Loops.* When data are collected in a list, we often want to perform the same operations on each element in the list. We then need to walk through all list elements. Computer languages have a special construct for doing this conveniently, and this construct is in Python and many other languages called a `for` loop. Let us use a `for` loop to print out all list elements:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print 'list element:', C
print 'The degrees list has', len(degrees), 'elements'
```

The `for C in degrees` construct creates a loop over all elements in the list `degrees`. In each pass of the loop, the variable `C` refers to an element in the list, starting with `degrees[0]`, proceeding with `degrees[1]`, and so on, before ending with the last element `degrees[n-1]` (if `n` denotes the number of elements in the list, `len(degrees)`).

The `for` loop specification ends with a colon, and after the colon comes a block of statements which does something useful with the current element. Each statement in the block must be indented, as we explained for `while` loops. In the example above, the block belonging to the `for` loop contains only one statement. The final `print` statement has the same indentation (none in this example) as the `for` statement and is executed as soon as the loop is terminated.

As already mentioned, understanding all details of a program by following the program flow by hand is often a very good idea. Here, we first define a list `degrees` containing 5 elements. Then we enter the `for` loop. In the first pass of the loop, `C` refers to the first element in the list `degrees`, i.e., the `int` object holding the value `0`. Inside the loop we then print out the text `'list element:'` and the value of `C`, which is `0`. There are no more statements in the loop block, so we proceed with the next pass of the loop. `C` then refers to the `int` object `10`, the output now prints `10` after the leading text, we proceed with `C` as the integers `20` and `40`, and finally `C` is `100`. After having printed the list element with value `100`, we move on to the statement after the indented loop block, which prints out the number of list elements. The total output becomes

```
list element: 0
list element: 10
list element: 20
list element: 40
list element: 100
The degrees list has 5 elements
```

Correct indentation of statements is crucial in Python, and we therefore strongly recommend you to work through Exercise 2.55 to learn more about this topic.

*Making the Table.* Our knowledge of lists and `for` loops over elements in lists puts us in a good position to write a program where we collect all the Celsius degrees to appear in the table in a list `Cdegrees`, and then use a `for` loop to compute and write out the corresponding Fahrenheit degrees. The complete program may look like this:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print C, F
```

The `print C, F` statement just prints the value of `C` and `F` with a default format, where each number is separated by one space character (blank). This does not look like a nice table (the output is identical to the one shown on page 52). Nice formatting is obtained by forcing `C` and `F` to be written in fields of fixed width and with a fixed number of decimals. An appropriate printf format is `%5d` (or `%5.0f`) for `C` and `%5.1f` for `F`. We may also add a headline to the table. The complete program becomes:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
print '    C    F'
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
```

This code is found in the file `c2f_table_list.py` and its output becomes

```
    C     F
  -20  -4.0
  -15   5.0
  -10  14.0
   -5  23.0
    0  32.0
    5  41.0
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
   35  95.0
   40 104.0
```

### 2.1.6 Alternative Implementations with Lists and Loops

We have already solved the problem of printing out a nice-looking conversion table for Celsius and Fahrenheit degrees. Nevertheless, there are usually many alternative ways to write a program that solves a specific problem. The next paragraphs explore some other possible Python constructs and programs to store numbers in lists and print out tables. The various code snippets are collected in the program file `c2f_table_lists.py`.

*While Loop Implementation of a For Loop.* Any `for` loop can be implemented as a `while` loop. The general code

```
for element in somelist:
    <process element>
```

can be transformed to this `while` loop:

```
index = 0
while index < len(somelist):
    element = somelist[index]
    <process element>
    index += 1
```

In particular, the example involving the printout of a table of Celsius and Fahrenheit degrees can be implemented as follows in terms of a `while` loop:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
print '    C    F'
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
    index += 1
```

*Storing the Table Columns as Lists.* A slight change of the previous program could be to store both the Celsius and Fahrenheit degrees in lists. For the Fahrenheit numbers we may start with an empty list `Fdegrees` and use `append` to add list elements inside the loop:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
Fdegrees = []       # start with empty list
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)
```

If we now print `Fdegrees` we get

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
 68.0, 77.0, 86.0, 95.0, 104.0]
```

*Loops with List Indices.* Instead of having a `for` loop over the list elements we may use a `for` loop over the list indices. The indices are integers going from 0 up to the length of the list minus one. Python has a `range` function returning such a list of integers:

- `range(n)` returns `[0, 1, 2, ..., n-1]`.
- `range(start, stop, step)` returns a list of `start`, `start+step`, `start+2*step`, and so on up to, but not including, `stop`. For example, `range(2, 8, 3)` returns `[2, 5]`, while `range(1, 11, 2)` returns `[1, 3, 5, 7, 9]`.
- `range(start, stop)` is the same as `range(start, stop, 1)`.

All legal indices of a list `a` are obtained by calling `range(len(a))`.

The previous `for` loop can alternatively make use of the `range` function and loops over list indices:

```
Cdegrees = range(-20, 45, 5)     # generate C values
Fdegrees = [0.0]*len(Cdegrees)   # list of 0.0 values
for i in range(len(Cdegrees)):
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
```

Observe that we need to initialize `Fdegrees` to be a list of length `len(Cdegrees)` (setting each element to zero for convenience). If we fail to let `Fdegrees` have the same length as `Cdegrees`, and set `Fdegrees`

`= []` instead, `Fdegrees[i]` will lead to an error messages saying that the index `i` is out of range. (Even index `0`, referring to the first element, is out of range if `Fdegrees` is an empty list.)

*Loops over Real Numbers.* So far, the data in `Cdegrees` have been integers. To make real numbers in the `Cdegrees` list, we cannot simply call the `range` function since it only generates integers. A loop is necessary for generating real numbers:

```
C_step = 0.5
C_start = -5
n = 16
Cdegrees = [0.0]*n;  Fdegrees = [0.0]*n
for i in range(n):
    Cdegrees[i] = C_start + i*C_step
    Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
```

A `while` loop with growing lists can also be used if we specify a stop value for `C`:

```
C_start = -5;  C_step = 0.5;  C_stop = 20
C = C_start
Cdegrees = [];  F_degrees = []
while C <= C_stop:
    Cdegrees.append(C)
    F = (9.0/5)*C + 32
    Fdegrees.append(F)
    C = C + C_step
```

*About Changing a List.* We have two seemingly alternative ways to traverse a list, either a loop over elements or over indices. Suppose we want to change the `Cdegrees` list by adding 5 to all elements. We could try

```
for c in Cdegrees:
    c += 5
```

but this loop leaves `Cdegrees` unchanged, while

```
for i in range(len(Cdegrees)):
    Cdegrees[i] += 5
```

works as intended. What is wrong with the first loop? The problem is that `c` is an ordinary variable which refers to a list element in the loop, but when we execute `c += 5`, we let `c` refer to a new `float` object (`c+5`). This object is never “inserted” in the list. The first two passes of the loop are equivalent to

```
c = Cdegrees[0]   # automatically done in the for statement
c += 5
c = Cdegrees[1]   # automatically done in the for statement
c += 5
```

The variable `c` can only be used to read list elements and never to change them. Only an assignment of the form

```
Cdegrees[i] = ...
```

can change a list element.

There is a way of traversing a list where we get both the index and an element in each pass of the loop:

```
for i, c in enumerate(Cdegrees):
    Cdegrees[i] = c + 5
```

This loop also adds 5 to all elements in the list.

*List Comprehension.* Because running through a list and for each element creating a new element in another list is a frequently encountered task, Python has a special compact syntax for doing this, called *list comprehension*. The general syntax reads

```
newlist = [E(e) for e in list]
```

where `E(e)` represents an expression involving element `e`. Here are three examples:

```
Cdegrees = [-5 + i*0.5 for i in range(n)]
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
C_plus_5 = [C+5 for C in Cdegrees]
```

List comprehensions are recognized as a `for` loop inside square brackets and will be frequently examplified thoughout the book.

*Traversing Multiple Lists Simultaneously.* We may use the `Cdegrees` and `Fdegrees` lists to make a table. To this end, we need to traverse both arrays. The `for element in list` construction is not suitable in this case, since it extracts elements from one list only. A solution is to use a `for` loop over the integer indices so that we can index both lists:

```
for i in range(len(Cdegrees)):
    print '%5d %5.1f' % (Cdegrees[i], Fdegrees[i])
```

It happens quite frequently that two or more lists need to be traversed simultaneously. As an alternative to the loop over indices, Python offers a special nice syntax that can be sketched as

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
    # work with element e1 from list1, element e2 from list2,
    # element e3 from list3, etc.
```

The `zip` function turns $n$ lists (`list1, list2, list3, ...`) into one list of $n$-tuples, where each $n$-tuple (`e1,e2,e3,...`) has its first element (`e1`) from the first list (`list1`), the second element (`e2`) from the second list (`list2`), and so forth. The loop stops when the end of the shortest list is reached. In our specific case of iterating over the two lists `Cdegrees` and `Fdegrees`, we can use the `zip` function:

```
for C, F in zip(Cdegrees, Fdegrees):
    print '%5d %5.1f' % (C, F)
```

It is considered more "Pythonic" to iterate over list elements, here `C` and `F`, rather than over list indices as in the `for i in range(len(Cdegrees))` construction.

### 2.1.7 Nested Lists

Our table data have so far used one separate list for each column. If there were $n$ columns, we would need $n$ list objects to represent the data in the table. However, we think of a table as *one* entity, not a collection of $n$ columns. It would therefore be natural to use one argument for the whole table. This is easy to achieve using a *nested list*, where each entry in the list is a list itself. A table object, for instance, is a list of lists, either a list of the row elements of the table or a list of the column elements of the table. Here is an example where the table is a list of two columns, and each column is a list of numbers[6]:

```
Cdegrees = range(-20, 41, 5)   # -20, -15, ..., 35, 40
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table = [Cdegrees, Fdegrees]
```

With the subscript `table[0]` we can access the first element (the `Cdegrees` list), and with `table[0][2]` we reach the third element in the list that constitutes the first element in `table` (this is the same as `Cdegrees[2]`).

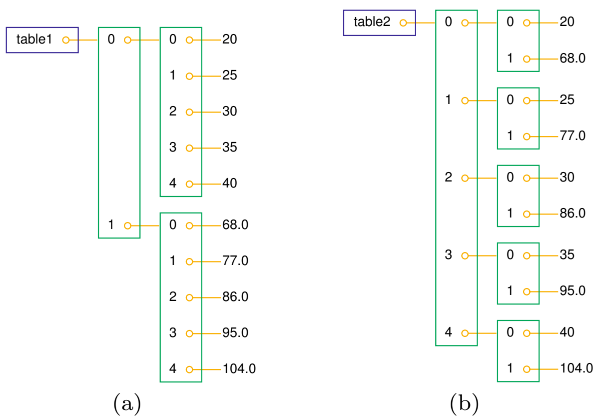



**Fig. 2.2** Two ways of creating a table as a nested list: (a) table of columns `C` and `F` (`C` and `F` are lists); (b) table of rows (`[C, F]` lists of two floats).

[6] Any value in $[41, 45]$ can be used as second argument (stop value) to `range` and will ensure that 40 is included in the range of generate numbers.

However, tabular data with rows and columns usually have the convention that the underlying data is a nested list where the first index counts the rows and the second index counts the columns. To have `table` on this form, we must construct `table` as a list of `[C, F]` pairs. The first index will then run over rows `[C, F]`. Here is how we may construct the nested list:

```
table = []
for C, F in zip(Cdegrees, Fdegrees):
    table.append([C, F])
```

We may shorten this code segment by introducing a list comprehension:

```
table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

This construction loops through pairs `C` and `F`, and for each pass in the loop we create a list element `[C, F]`.

The subscript `table[1]` refers to the second element in `table`, which is a `[C, F]` pair, while `table[1][0]` is the `C` value and `table[1][1]` is the `F` value. Figure 2.2 illustrates both a list of columns and a list of pairs. Using this figure, you can realize that the first index looks up the "main list", while the second index looks up the "sublist".

### 2.1.8 Printing Objects

*Modules for Pretty Print of Objects.* We may write `print table` to immediately view the nested list `table` from the previous section. In fact, any Python object `obj` can be printed to the screen by the command `print obj`. The output is usually one line, and this line may become very long if the list has many elements. For example, a long list like our `table` variable, demands a quite long line when printed.

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], ............, [40, 104.0]]
```

Splitting the output over several shorter lines makes the layout nicer and more readable. The `pprint` module offers a "pretty print" functionality for this purpose. The usage of `pprint` looks like

```
import pprint
pprint.pprint(table)
```

and the corresponding output becomes

```
[[-20, -4.0],
 [-15, 5.0],
 [-10, 14.0],
 [-5, 23.0],
 [0, 32.0],
 [5, 41.0],
 [10, 50.0],
 [15, 59.0],
 [20, 68.0],
```

```
[25, 77.0],
[30, 86.0],
[35, 95.0],
[40, 104.0]]
```

With this book comes a slightly modified `pprint` module having the name `scitools.pprint2`. This module allows full format control of the printing of the `float` objects in lists by specifying `scitools.pprint2.float_format` as a printf format string. The following example demonstrates how the output format of real numbers can be changed:

```
>>> import pprint, scitools.pprint2
>>> somelist = [15.8, [0.2, 1.7]]
>>> pprint.pprint(somelist)
[15.800000000000001, [0.20000000000000001, 1.7]]
>>> scitools.pprint2.pprint(somelist)
[15.8, [0.2, 1.7]]
>>> # default output is '%g', change this to
>>> scitools.pprint2.float_format = '%.2e'
>>> scitools.pprint2.pprint(somelist)
[1.58e+01, [2.00e-01, 1.70e+00]]
```

As can be seen from this session, the `pprint` module writes floating-point numbers with a lot of digits, in fact so many that we explicitly see the round-off errors. Many find this type of output is annoying and that the default output from the `scitools.pprint2` module is more like one would desire and expect.

The `pprint` and `scitools.pprint2` modules also have a function `pformat`, which works as the `pprint` function, but it returns a pretty formatted string rather than printing the string:

```
s = pprint.pformat(somelist)
print s
```

This last `print` statement prints the same as `pprint.pprint(somelist)`.

*Manual Printing.* Many will argue that tabular data such as those stored in the nested `table` list are not printed in a particularly pretty way by the `pprint` module. One would rather expect pretty output to be a table with two nicely aligned columns. To produce such output we need to code the formatting manually. This is quite easy: We loop over each row, extract the two elements `C` and `F` in each row, and print these in fixed-width fields using the printf syntax. The code goes as follows:

```
for C, F in table:
    print '%5d %5.1f' % (C, F)
```

### 2.1.9 Extracting Sublists

Python has a nice syntax for extracting parts of a list structure. Such parts are known as *sublists* or *slices*:

`A[i:]` is the sublist starting with index `i` in `A` and continuing to the end of `A`:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]
[8, 10]
```

`A[i:j]` is the sublist starting with index `i` in `A` and continuing up to and including index `j-1`. Make sure you remember that the element corresponding to index `j` is not included in the sublist:

```
>>> A[1:3]
[3.5, 8]
```

`A[:i]` is the sublist starting with index 0 in `A` and continuing up to and including the element with index `i-1`:

```
>>> A[:3]
[2, 3.5, 8]
```

`A[1:-1]` extracts all elements except the first and the last (recall that index `-1` refers to the last element), and `A[:]` is the whole list:

```
>>> A[1:-1]
[3.5, 8]
>>> A[:]
[2, 3.5, 8, 10]
```

In nested lists we may use slices in the first index:

```
>>> table[4:]
[[0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0],
 [25, 77.0], [30, 86.0], [35, 95.0], [40, 104.0]]
```

Sublists are always copies of the original list, so if you modify the sublist the original list remains unaltered and vice versa:

```
>>> l1 = [1, 4, 3]
>>> l2 = l1[:-1]
>>> l2
[1, 4]
>>> l1[0] = 100
>>> l1               # l1 is modified
[100, 4, 3]
>>> l2               # l2 is not modified
[1, 4]
```

The fact that slicing makes a copy can also be illustrated by the following code:

```
>>> B = A[:]
>>> C = A
>>> B == A
True
>>> B is A
False
>>> C is A
True
```

The `B == A` boolean expression is true if all elements in `B` are equal to the corresponding elements in `A`. The test `B is A` is true if `A` and `B` are names for the same list. Setting `C = A` makes `C` refer to the same list object as `A`, while `B = A[:]` makes `B` refer to a copy of the list referred to by `A`.

*Example.* We end this information on sublists by writing out the part of the `table` list of `[C, F]` rows (cf. Chapter 2.1.7) where the Celsius degrees are between 10 and 35 (not including 35):

```
>>> for C, F in table[Cdegrees.index(10):Cdegrees.index(35)]:
...     print '%5.0f %5.1f' % (C, F)
...
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

You should always stop reading and convince yourself that you understand why a code segment produces the printed output. In this latter example, `Cdegrees.index(10)` returns the index corresponding to the value `10` in the `Cdegrees` list. Looking at the `Cdegrees` elements, one realizes (do it!) that the `for` loop is equivalent to

```
for C, F in table[6:11]:
```

This loop runs over the indices $6, 7, \ldots, 10$ in `table`.

### 2.1.10 Traversing Nested Lists

We have seen that traversing the nested list `table` could be done by a loop of the form

```
for C, F in table:
    # process C and F
```

This is natural code when we know that `table` is a list of `[C, F]` lists. Now we shall address more general nested lists where we do not necessarily know how many elements there are in each list element of the list.

Suppose we use a nested list `scores` to record the scores of players in a game: `scores[i]` holds a list of the historical scores obtained by player number `i`. Different players have played the game a different number of times, so the length of `scores[i]` depends on `i`. Some code may help to make this clearer:

```
scores = []
# score of player no. 0:
scores.append([12, 16, 11, 12])
# score of player no. 1:
```

```
scores.append([9])
# score of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

The list `scores` has three elements, each element corresponding to a player. The element no. `g` in the list `scores[p]` corresponds to the score obtained in game number `g` played by player number `p`. The length of the lists `scores[p]` varies and equals 4, 1, and 8 for `p` equal to 0, 1, and 2, respectively.

In the general case we may have $n$ players, and some may have played the game a large number of times, making `scores` potentially a big nested list. How can we traverse the `scores` list and write it out in a table format with nicely formatted columns? Each row in the table corresponds to a player, while columns correspond to scores. For example, the data initialized above can be written out as

```
  12  16  11  12
   9
   6   9  11  14  17  15  14  20
```

In a program, we must use two *nested loops*, one for the elements in `scores` and one for the elements in the sublists of `scores`. The example below will make this clear.

There are two basic ways of traversing a nested list: either we use integer indices for each index, or we use variables for the list elements. Let us first exemplify the index-based version:

```
for p in range(len(scores)):
    for g in range(len(scores[p])):
        score = scores[p][g]
        print '%4d' % score,
    print
```

With the trailing comma after the print string, we avoid a newline so that the column values in the table (i.e., scores for one player) appear at the same line. The single `print` command after the loop over `c` adds a newline after each table row. The reader is encouraged to go through the loops by hand and simulate what happens in each statement (use the simple `scores` list initialized above).

The alternative version where we use variables for iterating over the elements in the `scores` list and its sublists looks like this:

```
for player in scores:
    for game in player:
        print '%4d' % game,
    print
```

Again, the reader should step through the code by hand and realize what the values of `player` and `game` are in each pass of the loops.

In the very general case we can have a nested list with many indices: `somelist[i1][i2][i3]...`. To visit each of the elements in the list, we use as many nested `for` loops as there are indices. With four indices, iterating over integer indices look as

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # work with value
```

The corresponding version iterating over sublists becomes

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # work with value
```

We recommend to do Exercise 2.58 to get a better understanding of nested `for` loops.

### 2.1.11 Tuples

Tuples are very similar to lists, but tuples cannot be changed. That is, a tuple can be viewed as a "constant list". While lists employ square brackets, tuples are written with standard parentheses:

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple with name t
```

One can also drop the parentheses in many occasions:

```
>>> t = 2, 4, 6, 'temp.pdf'
>>> for element in 'myfile.txt', 'yourfile.txt', 'herfile.txt':
...     print element,
...
myfile.txt yourfile.txt herfile.txt
```

The `for` loop here is over a tuple, because a comma separated sequence of objects, even without enclosing parentheses, becomes a tuple. Note the trailing comma in the `print` statement. This comma suppresses the final newline that the `print` command automatically adds to the output string. This is the way to make several `print` statements build up one line of output.

Much functionality for lists is also available for tuples, for example:

```
>>> t = t + (-1.0, -2.0)           # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                           # indexing
4
>>> t[2:]                          # subtuple/slice
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                         # membership
True
```

Any list operation that changes the list will not work for tuples:

```
>>> t[1] = -1
...
TypeError: object does not support item assignment

>>> t.append(0)
...
AttributeError: 'tuple' object has no attribute 'append'

>>> del t[1]
...
TypeError: object doesn't support item deletion
```

Some list methods, like `index`, are not available for tuples.

So why do we need tuples when lists can do more than tuples?

- Tuples protect against accidental changes of their contents.
- Code based on tuples is faster than code based on lists.
- Tuples are frequently used in Python software that you certainly will make use of, so you need to know this data type.

There is also a fourth argument, which is important for a data type called dictionaries (introduced in Chapter 6.2): tuples can be used as keys in dictionaries while lists can not.

## 2.2 Functions

In a computer language like Python, the term *function* means more than just a mathematical function. A function is a collection of statements that you can execute wherever and whenever you want in the program. You may send variables to the function to influence what is getting computed by statements in the function, and the function may return new objects. In particular, functions help to avoid duplicating code snippets by putting all similar snippets in a common place. This strategy saves typing and makes it easier to change the program later. Functions are also often used to just split a long program into smaller, more managable pieces, so the program and your own thinking about it become clearer. Python comes with lots of functions (`math.sqrt`, `range`, and `len` are examples we have met so far). This section outlines how you can define your own functions.

### 2.2.1 Functions of One Variable

Let us start with making a Python function that evaluates a mathematical function, more precisely the function $F(C)$ defined in (1.2): $F(C) = \frac{9}{5}C + 32$. The corresponding Python function must take $C$ as argument and return the value $F(C)$. The code for this looks like

```
def F(C):
    return (9.0/5)*C + 32
```

All Python functions begin with `def`, followed by the function name, and then inside parentheses a comma-separated list of *function arguments*. Here we have only one argument `C`. This argument acts as a standard variable inside the function. The statements to be performed inside the function must be indented. At the end of a function it is common to *return* a value, that is, send a value "out of the function". This value is normally associated with the name of the function, as in the present case where the returned value is $F(C)$.

The `def` line with the function name and arguments is often referred to as the *function header*, while the indented statements constitute the *function body*.

To use a function, we must *call*[7] it. Because the function returns a value, we need to store this value in a variable or make use of it in other ways. Here are some calls to `F`:

```
a = 10
F1 = F(a)
temp = F(15.5)
print F(a+1)
sum_temp = F(10) + F(20)
```

The returned object from `F(C)` is in our case a `float` object. The call `F(C)` can therefore be placed anywhere in a code where a `float` object would be valid. The `print` statement above is one example. As another example, say we have a list `Cdegrees` of Celsius degrees and we want to compute a list of the corresponding Fahrenheit degrees using the `F` function above in a list comprehension:

```
Fdegrees = [F(C) for C in Cdegrees]
```

As an example of a slight variation of our `F(C)` function, we may return a formatted string instead of a real number:

```
>>> def F2(C):
...     F_value = (9.0/5)*C + 32
...     return '%.1f degrees Celsius corresponds to '\
...            '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> s1 = F2(21)
>>> s1
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
```

The assignment to `F_value` demonstrates that we can create variables inside a function as needed.

[7] Sometimes the word *invoke* is used as an alternative to *call*.

### 2.2.2 Local and Global Variables

Let us reconsider the F2(C) function from the previous section. The variable F_value is a *local* variable in the function, and a local variable does not exist outside the function. We can easily demonstrate this fact by continuing the previous interactive session:

```
>>> c1 = 37.5
>>> s2 = F2(c1)
>>> F_value
...
NameError: name 'F_value' is not defined
```

The surrounding program outside the function is not aware of F_value. Also the argument to the function, C, is a local variable that we cannot access outside the function:

```
>>> C
...
NameError: name 'C' is not defined
```

On the contrary, the variables defined outside of the function, like s1, s2, and c1 in the above session, are *global* variables. These can be accessed everywhere in a program.

Local variables are created inside a function and destroyed when we leave the function. To learn more about this fact, we may study the following session where we write out F_value, C, and some global variable r inside the function:

```
>>> def F3(C):
...     F_value = (9.0/5)*C + 32
...     print 'Inside F3: C=%s F_value=%s r=%s' % (C, F_value, r)
...     return '%.1f degrees Celsius corresponds to '\
...            '%.1f degrees Fahrenheit' % (C, F_value)
...
>>> C = 60    # make a global variable C
>>> r = 21    # another global variable
>>> s3 = F3(r)
Inside F3: C=21 F_value=69.8 r=21
>>> s3
'21.0 degrees Celsius corresponds to 69.8 degrees Fahrenheit'
>>> C
60
```

This example illustrates that there are two C variables, one global, defined in the main program with the value 60 (an int object), and one local, living when the program flow is inside the F3 function. The value of this C is given in the call to the F3 function (also an int object in this case). Inside the F3 function the local C "hides" the global C variable in the sense that when we refer to C we access the local variable[8].

The more general rule, when you have several variables with the same name, is that Python first tries to look up the variable name

[8] The global C can technically be accessed as globals()['C'], but one should avoid working with local and global variables with the same names at the same time!

among the local variables, then there is a search among global variables, and finally among built-in Python functions. Here is a complete sample program with several versions of a variable `sum` which aims to illustrate this rule:

```
print sum  # sum is a built-in Python function
sum = 500  # rebind the name sum to an int
print sum  # sum is a global variable

def myfunc(n):
    sum = n + 1
    print sum  # sum is a local variable
    return sum

sum = myfunc(2) + 1   # new value in global variable sum
print sum
```

In the first line, there are no local variables, so Python searches for a global value with name `sum`, but cannot find any, so the search proceeds with the built-in functions, and among them Python finds a function with name `sum`. The printout of `sum` becomes something like `<built-in function sum>`.

The second line rebinds the global name `sum` to an `int` object. When trying to access `sum` in the next `print` statement, Python searches among the global variables (no local variables so far) and finds one. The printout becomes `500`. The call `myfunc(2)` invokes a function where `sum` is a local variable. Doing a `print sum` in this function makes Python first search among the local variables, and since `sum` is found there, the printout becomes `3` (and not `500`, the value of the global variable `sum`). The value of the local variable `sum` is returned, added to `1`, to form an `int` object with value `4`. This `int` object is then bound to the global variable `sum`. The final `print sum` leads to a search among global variables, and we find one with value `4`.

The values of global variables can be accessed inside functions, but the values cannot be changed unless the variable is declared as `global`:

```
a = 20; b = -2.5        # global variables

def f1(x):
    a = 21              # this is a new local variable
    return a*x + b      # 21*x - 2.5

print a                 # yields 20

def f2(x):
    global a
    a = 21              # the global a is changed
    return a*x + b      # 21*x - 2.5

f1(3); print a          # 20 is printed
f2(3); print a          # 21 is printed
```

Note that in the `f1` function, `a = 21` creates a local variable `a`. As a programmer you may think you change the global `a`, but it does not

happen! Normally, this feature is advantageous because changing global variables often leads to errors in programs.

### 2.2.3 Multiple Arguments

The previous `F(C)` and `F2(C)` functions are functions of one variable, `C`, or as we phrase it in computer science: the functions take one argument (`C`). Functions can have as many arguments as desired; just separate the argument names by commas.

Consider the function $y(t)$ in (1.1). Here is a possible Python function taking two arguments:

```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

Note that `g` is a local variable with a fixed value, while `t` and `v0` are arguments and therefore also local variables. Examples on valid calls are

```
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)
```

The possibility to write `argument=value` in the call makes it easier to read and understand the call statement. With the `argument=value` syntax for all arguments, the sequence of the arguments does not matter in the call, which here means that we may put `v0` before `t`. When omitting the `argument=` part, the sequence of arguments in the call must perfectly match the sequence of arguments in the function definition. The `argument=value` arguments must appear after all the arguments where only `value` is provided (e.g., `yfunc(t=0.1, 6)` is illegal).

Whether we write `yfunc(0.1, 6)` or `yfunc(v0=6, t=0.1)`, the arguments are initialized as local variables in the function in the same way as when we assign values to variables:

```
    t = 0.1
    v0 = 6
```

These statements are not visible in the code, but a call to a function automatically initializes the arguments in this way.

Some may argue that `yfunc` should be a function of `t` only, because we mathematically think of $y$ as a function of $t$ and write $y(t)$. This is easy to reflect in Python:

```
def yfunc(t):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

The main difference is that `v0` now must be a *global* variable, which needs to be initialized before we call `yfunc`. The next session demonstrates what happens if we fail to initialize such a global variable:

```
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
...
>>> yfunc(0.6)
...
NameError: global name 'v0' is not defined
```

The remedy is to define `v0` as a global variable prior to calling `yfunc`:

```
>>> v0 = 5
>>> yfunc(0.6)
1.2342
```

So far our Python functions have typically computed some mathematical function, but the usefulness of Python functions goes far beyond mathematical functions. Any set of statements that we want to repeatedly execute under slightly different circumstances is a candidate for a Python function. Say we want to make a list of numbers starting from some value and stopping at another value, with increments of a given size. With corresponding variables `start=2`, `stop=8`, and `inc=2`, we should produce the numbers 2, 4, 6, and 8. Our tables in this chapter typically needs such functionality for creating a list of $C$ values or a list of $t$ values. Let us therefore write a function doing the task[9], together with a couple of statements that demonstrate how we call the function:

```
def makelist(start, stop, inc):
    value = start
    result = []
    while value <= stop:
        result.append(value)
        value = value + inc
    return result

mylist = makelist(0, 100, 0.2)
print mylist  # will print 0, 0.2, 0.4, 0.6, ... 99.8, 100
```

The `makelist` function has three arguments: `start`, `stop`, and `inc`, which become local variables in the function. Also `value` and `result` are local variables. In the surrounding program we define only one variable, `mylist`, and this is then a global variable.

[9] You might think that `range(start, stop, inc)` makes the `makelist` function redundant, but `range` can only generate integers, while `makelist` can generate real numbers too – and more, see Exercise 2.40.

### 2.2.4 Multiple Return Values

Python functions may return more than one value. Suppose we are interested in evaluating both $y(t)$ defined in (1.1) and its derivative

$$\frac{dy}{dt} = v_0 - gt\,.$$

In the current application, $y'(t)$ has the physical interpretation as the velocity of the ball. To return $y$ and $y'$ we simply separate their corresponding variables by a comma in the `return` statement:

```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
```

When we call this latter `yfunc` function, we need two values on the left-hand side of the assignment operator because the function returns two values:

```
position, velocity = yfunc(0.6, 3)
```

Here is an application of the `yfunc` function for producing a nicely formatted table of positions and velocities of a ball thrown up in the air:

```
t_values = [0.05*i for i in range(10)]
for t in t_values:
    pos, vel = yfunc(t, v0=5)
    print 't=%-10g position=%-10g velocity=%-10g' % (t, pos, vel)
```

The format `%-10g` prints a real number as compactly as possible (decimal or scientific notation) in a field of width 10 characters. The minus ("-") sign after the percentage sign implies that the number is *left-adjusted* in this field, a feature that is important for creating nice-looking columns in the output:

```
t=0          position=0          velocity=5
t=0.05       position=0.237737   velocity=4.5095
t=0.1        position=0.45095    velocity=4.019
t=0.15       position=0.639638   velocity=3.5285
t=0.2        position=0.8038     velocity=3.038
t=0.25       position=0.943437   velocity=2.5475
t=0.3        position=1.05855    velocity=2.057
t=0.35       position=1.14914    velocity=1.5665
t=0.4        position=1.2152     velocity=1.076
t=0.45       position=1.25674    velocity=0.5855
```

When a function returns multiple values, separated by a comma in the `return` statement, a tuple (Chapter 2.1.11) is actually returned. We can demonstrate that fact by the following session:

```
>>> def f(x):
...     return x, x**2, x**4
...
```

```
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)
```

Note that storing multiple return values into separate variables, as we do in the last line, is actually the same functionality as we use for storing list elements in separate variables, see on page 58.

Our next example concerns a function aimed at calculating the sum

$$L(x; n) = \sum_{i=1}^{n} \frac{1}{i} \left( \frac{x}{1+x} \right)^{i} . \tag{2.1}$$

It can be shown that $L(x; n)$ is an approximation to $\ln(1 + x)$ for a finite $n$ and $x \geq 1$. The approximation becomes exact in the limit:

$$\ln(1 + x) = \lim_{n\to\infty} L(x; n) .$$

To compute a sum in a Python program, we use a loop and add terms to a "summing variable" inside the loop. This variable must be initialized to zero outside the loop. For example, we can sketch the implementation of $\sum_{i=1}^{n} c(i)$, where $c(i)$ is some formula depending on $i$, as

```
s = 0
for i in range(1, n+1):
    s += c(i)
```

For the specific sum (2.1) we just replace `c(i)` by the right term $(1/i)(x/(1 + x))^i$ inside the `for` loop[10]:

```
s = 0
for i in range(1, n+1):
    s += (1.0/i)*(x/(1.0+x))**i
```

It is natural to embed the computation of the sum in a function which takes $x$ and $n$ as arguments and returns the sum:

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    return s
```

Instead of just returning the value of the sum, we could return additional information on the error involved in the approximation of $\ln(1 + x)$ by $L(x; n)$. The first neglected term in the sum provides

[10] Observe the `1.0` numbers: These avoid integer division (`i` is `int` and `x` may be `int`).

an indication of the error[11]. We could also return the exact error. The new version of the L(x, n) function then looks as this:

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
value, approximate_error, exact_error = L2(x, 100)
```

The next section demonstrates the usage of the L function to judge the quality of the approximation $L(x;n)$ to $\ln(1+x)$.

### 2.2.5 Functions with No Return Values

Sometimes a function just performs a set of statements, and it is not natural to return any values to the calling code. In such situations one can simply skip the return statement. Some programming languages use the terms procedure or subroutine for functions that do not return anything.

Let us exemplify a function without return values by making a table of the accuracy of the $L(x;n)$ approximation to $\ln(1+x)$ from the previous section:

```
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L(x, n)
        print 'n=%-4d %-10g  (next term: %8.2e  '\
              'error: %8.2e)' % (n, value, next, error)
```

This function just performs a set of statements that we may want to run several times. Calling

```
table(10)
table(1000)
```

gives the output :

```
x=10, ln(1+x)=2.3979
n=1    0.909091    (next term: 4.13e-01  error: 1.49e+00)
n=2    1.32231     (next term: 2.50e-01  error: 1.08e+00)
n=10   2.17907     (next term: 3.19e-02  error: 2.19e-01)
n=100  2.39789     (next term: 6.53e-07  error: 6.59e-06)
n=500  2.3979      (next term: 3.65e-24  error: 6.22e-15)
```

[11] The size of the terms decreases with increasing $n$, and the first neglected term is then bigger than all the remaining terms, but not necessarily bigger than their sum. The first neglected term is therefore only an indication of the size of the total error we make.

```
x=1000, ln(1+x)=6.90875
n=1    0.999001    (next term: 4.99e-01  error: 5.91e+00)
n=2    1.498       (next term: 3.32e-01  error: 5.41e+00)
n=10   2.919       (next term: 8.99e-02  error: 3.99e+00)
n=100  5.08989     (next term: 8.95e-03  error: 1.82e+00)
n=500  6.34928     (next term: 1.21e-03  error: 5.59e-01)
```

From this output we see that the sum converges much more slowly when $x$ is large than when $x$ is small. We also see that the error is an order of magnitude or more larger than the first neglected term in the sum. The functions `L` and `table` are found in the file `lnsum.py`.

When there is no explicit `return` statement in a function, Python actually inserts an invisible `return None` statement. `None` is a special object in Python that represents something we might think of as "empty data" or "nothing". Other computer languages, such as C, C++, and Java, use the word "void" for a similar thing. Normally, one will call the `table` function without assigning the return value to any variable, but if we assign the return value to a variable, `result = table(500)`, `result` will refer to a `None` object.

The `None` value is often used for variables that should exist in a program, but where it is natural to think of the value as conceptually undefined. The standard way to test if an object `obj` is set to `None` or not reads

```
if obj is None:
    ...
if obj is not None:
    ...
```

One can also use `obj == None`. The `is` operator tests if two names refer to the same object, while `==` tests if the contents of two objects are the same:

```
>>> a = 1
>>> b = a
>>> a is b   # a and b refer to the same object
True
>>> c = 1.0
>>> a is c
False
>>> a == c   # a and c are mathematically equal
True
```

### 2.2.6 Keyword Arguments

Some function arguments can be given a default value so that we may leave out these arguments in the call, if desired. A typical function may look as

```
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>>     print arg1, arg2, kwarg1, kwarg2
```

The first two arguments, `arg1` and `arg2`, are *ordinary* or *positional* arguments, while the latter two are *keyword arguments* or *named arguments*. Each keyword argument has a name (in this example `kwarg1` and `kwarg2`) and an associated default value. The keyword arguments must always be listed after the positional arguments in the function definition.

When calling `somefunc`, we may leave out some or all of the keyword arguments. Keyword arguments that do not appear in the call get their values from the specified default values. We can demonstrate the effect through some calls:

```
>>> somefunc('Hello', [1,2])
Hello [1, 2] True 0
>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0
>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi
>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi
```

The sequence of the keyword arguments does not matter in the call. We may also mix the positional and keyword arguments if we explicitly write `name=value` for all arguments in the call:

```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[1,2],)
Hi [1, 2] 6 Hello
```

*Example: Function with Default Parameters.* Consider a function of $t$ which also contains some parameters, here $A$, $a$, and $\omega$:

$$f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)\,. \tag{2.2}$$

We can implement $f$ as a Python function where the independent variable $t$ is an ordinary positional argument, and the parameters $A$, $a$, and $\omega$ are keyword arguments with suitable default values:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)
```

Calling `f` with just the `t` argument specified is possible:

```
v1 = f(0.2)
```

In this case we evaluate the expression $e^{-0.2}\sin(2\pi\cdot 0.2)$. Other possible calls include

```
v2 = f(0.2, omega=1)
v3 = f(1, A=5, omega=pi, a=pi**2)
v4 = f(A=5, a=2, t=0.01, omega=0.1)
v5 = f(0.2, 0.5, 1, 1)
```

You should write down the mathematical expressions that arise from these four calls. Also observe in the third line above that a positional argument, `t` in that case, can appear in between the keyword arguments if we write the positional argument on the keyword argument form `name=value`. In the last line we demonstrate that keyword arguments can be used as positional argument, i.e., the name part can be skipped, but then the sequence of the keyword arguments in the call must match the sequence in the function definition exactly.

*Example: Computing a Sum with Default Tolerance.* Consider the $L(x; n)$ sum and the Python implementation `L(x, n)` from Chapter 2.2.4. Instead of specifying the number of terms in the sum, $n$, it is better to specify a tolerence $\varepsilon$ of the accuracy. We can use the first neglected term as an estimate of the accuracy. This means that we sum up terms as long as the absolute value of the next term is greater than $\epsilon$. It is natural to provide a default value for $\epsilon$:

```
def L2(x, epsilon=1.0E-6):
    x = float(x)
    i = 1
    term = (1.0/i)*(x/(1+x))**i
    s = term
    while abs(term) > epsilon:   # abs(x) is |x|
        i += 1
        term = (1.0/i)*(x/(1+x))**i
        s += term
    return s, i
```

Here is an example involving this function to make a table of the approximation error as $\epsilon$ decreases:

```
from math import log
x = 10
for k in range(4, 14, 2):
    epsilon = 10**(-k)
    approx, n = L2(x, epsilon=epsilon)
    exact = log(1+x)
    exact_error = exact - approx
    print 'epsilon: %5.0e, exact error: %8.2e, n=%d' % \
          (epsilon, exact_error, n)
```

The output becomes

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

We see that the `epsilon` estimate is almost 10 times smaller than the exact error, regardless of the size of `epsilon`. Since `epsilon` follows the exact error quite well over many orders of magnitude, we may view `epsilon` as a useful indication of the size of the error.

### 2.2.7 Doc Strings

There is a convention in Python to insert a documentation string right after the `def` line of the function definition. The documentation string, known as a *doc string*, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Doc strings are usually enclosed in triple double quotes `"""`, which allow the string to span several lines.

Here are two examples on short and long doc strings:

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: coefficients a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Note that the doc string must appear before any statement in the function body.

There are several Python tools that can automatically extract doc strings from the source code and produce various types of documentation, see [5, App. B.2]. The doc string can be accessed in a code as `funcname.__doc__`, where `funcname` is the name of the function, e.g.,

```
print line.__doc__
```

which prints out the documentation of the `line` function above:

```
Compute the coefficients a and b in the mathematical
expression for a straight line y = a*x + b that goes
through two points (x0, y0) and (x1, y1).

x0, y0: a point on the line (float objects).
x1, y1: another point on the line (float objects).
return: coefficients a, b for the line (y=a*x+b).
```

Doc strings often contain interactive sessions, copied from a Python shell, to illustrate how the function is used. We can add such a session to the doc string in the `line` function:

```
def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0,y0) and (x1,y1).
```

```
    x0, y0: a point on the line (float).
    x1, y1: another point on the line (float).
    return: coefficients a, b (floats) for the line (y=a*x+b).

    Example:
    >>> a, b = line(1, -1, 4, 3)
    >>> a
    1.3333333333333333
    >>> b
    -2.333333333333333
    """
    a = (y1 - y0)/float(x1 - x0)
    b = y0 - a*x0
    return a, b
```

A particularly nice feature is that all such interactive sessions in doc strings can be automatically run, and new results are compared to the results found in the doc strings. This makes it possible to use interactive sessions in doc strings both for exemplifying how the code is used and for testing that the code works.

### 2.2.8 Function Input and Output

It is a convention in Python that function arguments represent the input data to the function, while the returned objects represent the output data. We can sketch a general Python function as

```
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io6; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

Here `i1`, `i2`, `i3` are positional arguments representing input data; `io4` and `io5` are positional arguments representing input *and* output data; `i6` and `io7` are keyword arguments representing input and input/output data, respectively; and `o1`, `o2`, and `o3` are computed objects in the function, representing output data together with `io4`, `io5`, and `io7`. All examples later in the book will make use of this convention.

### 2.2.9 Functions as Arguments to Functions

Programs doing calculus frequently need to have functions as arguments in other functions. For example, for a mathematical function $f(x)$ we can have Python functions for

1. numerical root finding: solve $f(x) = 0$ approximately (Chapters 3.6.2 and 5.1.9)
2. numerical differentiation: compute $f'(x)$ approximately (Appendix A and Chapters 7.3.2 and 9.2)
3. numerical integration: compute $\int_a^b f(x)dx$ approximately (Appendix A and Chapters 7.3.3 and 9.3)

4. numerical solution of differential equations: $\frac{dx}{dt} = f(x)$ (Appendix B and Chapters 7.4 and 9.4)

In such Python functions we need to have the $f(x)$ function as an argument `f`. This is straightforward in Python and hardly needs any explanation, but in most other languages special constructions must be used for transferring a function to another function as argument.

As an example, consider a function for computing the second-order derivative of a function $f(x)$ numerically:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}, \tag{2.3}$$

where $h$ is a small number. The approximation (2.3) becomes exact in the limit $h \to 0$. A Python function for computing (2.3) can be implemented as follows:

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

The `f` argument is like any other argument, i.e., a name for an object, here a function object that we can call as we normally call function objects. An application of `diff2` can read

```
def g(t):
    return t**(-6)

t = 1.2
d2g = diff2(g, t)
print "g''(%f)=%f" % (t, d2g)
```

*The Behaviour of the Numerical Derivative as $h \to 0$.* From mathematics we know that the approximation formula (2.3) becomes more accurate as $h$ decreases. Let us try to demonstrate this expected feature by making a table of the second-order derivative of $g(t) = t^{-6}$ at $t = 1$ as $h \to 0$:

```
for k in range(1,15):
    h = 10**(-k)
    d2g = diff2(g, 1, h)
    print 'h=%.0e: %.5f' % (h, d2g)
```

The output becomes

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
```

```
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```

With $g(t) = t^{-6}$, the exact answer is $g''(1) = 42$, but for $h < 10^{-8}$ the computations give totally wrong answers! The problem is that for small $h$ on a computer, round-off errors in the formula (2.3) blow up and destroy the accuracy. The mathematical result that (2.3) becomes an increasingly better approximation as $h$ gets smaller and smaller does not hold on a computer! Or more precisely, the result holds until $h$ in the present case reaches $10^{-4}$.

The reason for the inaccuracy is that the numerator in (2.3) for $g(t) = t^{-6}$ and $t = 1$ contains subtraction of quantities that are almost equal. The result is a very small and inaccurate number. The inaccuracy is magnified by $h^{-2}$, a number that becomes very large for small $h$. Swithing from the standard floating-point numbers (`float`) to numbers with arbitrary high precision resolves the problem. Python has a module `decimal` that can be used for this purpose. The file `highprecision.py` solves the current problem using arithmetics based on the `decimal` module. With 25 digits in `x` and `h` inside the `diff2` function, we get accurate results for $h \leq 10^{-13}$. However, for most practical applications of (2.3), a moderately small $h$, say $10^{-3} \leq h \leq 10^{-4}$, gives sufficient accuracy and then round-off errors from `float` calculations do not pose problems. Real-world science or engineering applications usually have many parameters with uncertainty, making the end result also uncertain, and formulas like (2.3) can then be computed with moderate accuracy without affecting the overall computational error.

### 2.2.10 The Main Program

In programs containing functions we often refer to a part of the program that is called the *main program*. This is the collection of all the statements outside the functions, plus the definition of all functions. Let us look at a complete program:

```
from math import *                      # in main

def f(x):                               # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2                                   # in main
y = f(x)                                # in main
print 'f(%g)=%g' % (x, y)               # in main
```

The main program here consists of the lines with a comment `in main`. The execution always starts with the first line in the main program. When a function is encountered, its statements are just used to define the function – nothing gets computed inside the function before

we explicitly call the function, either from the main program or from another function. All variables initialized in the main program become global variables (see Chapter 2.2.2).

The program flow in the program above goes as follows:

1. Import functions from the `math` module,
2. define a function `f(x)`,
3. define `x`,
4. call `f` and execute the function body,
5. define `y` as the value returned from `f`,
6. print the string.

In point 4, we jump to the `f` function and execute the statement inside that function for the first time. Then we jump back to the main program and assign the `float` object returned from `f` to the `y` variable.

More information on program flow and the jump between the main program and functions is covered in Chapter 2.4.2 and Appendix D.1.

### 2.2.11 Lambda Functions

There is a quick one-line construction of functions that is sometimes convenient:

```
f = lambda x: x**2 + 4
```

This so-called *lambda function* is equivalent to writing

```
def f(x):
    return x**2 + 4
```

In general,

```
def g(arg1, arg2, arg3, ...):
    return expression
```

can be written as

```
g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions are usually used to quickly define a function as argument to another function. Consider, as an example, the `diff2` function from Chapter 2.2.9. In the example from that chapter we want to differentiate $g(t) = t^{-6}$ twice and first make a Python function `g(t)` and then send this `g` to `diff2` as argument. We can skip the step with defining the `g(t)` function and instead insert a lambda function as the `f` argument in the call to `diff2`:

```
d2 = diff2(lambda t: t**(-6), 1, h=1E-4)
```

Because lambda functions can be defined "on the fly" and thereby save typing of a separate function with `def` and an intended block, lambda functions are popular among many programmers.

Lambda functions may also take keyword arguments. For example,

```
d2 = diff2(lambda t, A=1, a=0.5: -a*2*t*A*exp(-a*t**2), 1.2)
```

## 2.3 If Tests

The flow of computer programs often needs to branch. That is, if a condition is met, we do one thing, and if not, we do another thing. A simple example is a function defined as

$$f(x) = \begin{cases} \sin x, & 0 \le x \le \pi \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

In a Python implementation of this function we need to test on the value of $x$, which can be done as displayed below:

```
def f(x):
    if 0 <= x <= pi:
        value = sin(x)
    else:
        value = 0
    return value
```

The general structure of an `if-else` test is

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

When `condition` evaluates to true, the program flow branches into the first block of statements. If `condition` is `False`, the program flow jumps to the second block of statements, after the `else:` line. As with `while` and `for` loops, the block of statements are indented. Here is another example:

```
if C < -273.15:
    print '%g degrees Celsius is non-physical!' % C
    print 'The Fahrenheit temperature will not be computed.'
else:
    F = 9.0/5*C + 32
    print F
print 'end of program'
```

The two `print` statements in the `if` block are executed if and only if `C < -273.15` evaluates to `True`. Otherwise, we jump over the first two `print` statements and carry out the computation and printing of `F`. The printout of `end of program` will be performed regardless of the outcome of the `if` test since this statement is not indented and hence neither a part of the `if` block nor the `else` block.

The `else` part of an `if` test can be skipped, if desired:

```
if condition:
    <block of statements>
<next statement>
```

For example,

```
if C < -273.15:
    print '%s degrees Celsius is non-physical!' % C
F = 9.0/5*C + 32
```

In this case the computation of `F` will always be carried out, since the statement is not indented and hence not a part of the `if` block.

With the keyword `elif`, short for *else if*, we can have several mutually exclusive `if` tests, which allows for multiple branching of the program flow:

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

The last `else` part can be skipped if it is not needed. To illustrate multiple branching we will implement a "hat" function, which is widely used in advanced computer simulations in science and industry. One example of a "hat" function is

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x < 1 \\ 2 - x, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases} \tag{2.5}$$

The solid line in Figure 4.11 on page 203 illustrates the shape of this function. The Python implementation associated with (2.5) needs multiple `if` branches:

```
def N(x):
    if x < 0:
        return 0.0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0.0
```

This code corresponds directly to the mathematical specification, which is a sound strategy that usually leads to fewer errors in programs. We could mention that there is another way of constructing this `if` test that results in shorter code:

```
def N(x):
    if 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    else:
        return 0
```

As a part of learning to program, understanding this latter sample code is important, but we recommend the former solution because of its direct similarity with the mathematical definition of the function.

A popular programming rule is to avoid multiple `return` statements in a function – there should only be one `return` at the end. We can do that in the `N` function by introducing a local variable, assigning values to this variable in the blocks and returning the variable at the end. However, we do not think an extra variable and an extra line make a great improvement in such a short function. Nevertheless, in long and complicated functions the rule can be helpful.

A variable is often assigned a value that depends on a boolean expression. This can be coded using a common `if-else` test:

```
if condition:
    a = value1
else:
    a = value2
```

Because this construction is often needed, Python provides a one-line syntax for the four lines above:

```
a = (value1 if condition else value2)
```

The parentheses are not required, but recommended style. One example is

```
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

Since the inline `if` test is an expression with a value, it can be used in lambda functions:

```
f = lambda x: sin(x) if 0 <= x <= 2*pi else 0
```

The traditional `if-else` construction with indented blocks cannot be used inside lambda functions because it is not just an expression (lambda functions cannot have statements inside them, only a single expression).

## 2.4 Summary

### 2.4.1 Chapter Topics

*While Loops.* Loops are used to repeat a collection of program statements several times. The statements that belong to the loop must be consistently indented in Python. A `while` loop runs as long as a condition evaluates to `True`:

```
>>> t = 0; dt = 0.5; T = 2
>>> while t <= T:
...     print t
...     t += dt
...
0
0.5
1.0
1.5
2.0
>>> print 'Final t:', t, '; t <= T is', t <= T
Final t: 2.5 ; t <= T is False
```

*Lists.* A list is used to collect a number of values or variables in an ordered sequence.

```
>>> mylist = [t, dt, T, 'mynumbers.dat', 100]
```

A list element can be any Python object, including numbers, strings, functions, and other lists, for instance. Table 2.1 shows some important list operations (only a subset of these are explained in the present chapter).

*Tuples.* A tuple can be viewed as a constant list: no changes in the contents of the tuple is allowed. Tuples employ standard parentheses or no parentheses, and elements are separated with comma as in lists:

```
>>> mytuple = (t, dt, T, 'mynumbers.dat', 100)
>>> mytuple =  t, dt, T, 'mynumbers.dat', 100
```

Many list operations are also valid for tuples. In Table 2.1, all operations can be applied to a tuple `a`, except those involving `append`, `del`, `remove`, `index`, and `sort`.

An object `a` containing an ordered collection of other objects such that `a[i]` refers to object with index `i` in the collection, is known as a *sequence* in Python. Lists, tuples, strings, and arrays (Chapter 4) are examples on sequences. You choose a sequence type when there is a natural ordering of elements. For a collection of unordered objects a *dictionary* (introduced in Chapter 6.2) is often more convenient.

*For Loops.* A `for` loop is used to run through the elements of a list or a tuple:

**Table 2.1** Summary of important functionality for list objects.

| | |
|---|---|
| `a = []` | initialize an empty list |
| `a = [1, 4.4, 'run.py']` | initialize a list |
| `a.append(elem)` | add `elem` object to the end |
| `a + [1,3]` | add two lists |
| `a.insert(i, e)` | insert element `e` before index `i` |
| `a[3]` | index a list element |
| `a[-1]` | get last list element |
| `a[1:3]` | slice: copy data to sublist (here: index 1, 2) |
| `del a[3]` | delete an element (index `3`) |
| `a.remove(e)` | remove an element with value `e` |
| `a.index('run.py')` | find index corresponding to an element's value |
| `'run.py' in a` | test if a value is contained in the list |
| `a.count(v)` | count how many elements that have the value `v` |
| `len(a)` | number of elements in list `a` |
| `min(a)` | the smallest element in `a` |
| `max(a)` | the largest element in `a` |
| `sum(a)` | add all elements in `a` |
| `as = sorted(a)` | sort list `a` (return new list) |
| `sorted(a)` | return sorted version of list `a` |
| `reverse(a)` | return reversed sorted version of list `a` |
| `b[3][0][2]` | nested list indexing |
| `isinstance(a, list)` | is `True` if `a` is a list |

```
>>> for elem in [10, 20, 25, 27, 28.5]:
...     print elem,
...
10 20 25 27 28.5
```

The trailing comma after the `print` statement prevents the newline character which `print` otherwise adds to the character.

The `range` function is frequently used in `for` loops over a sequence of integers. Recall that `range(start, stop, inc)` does not include the "end value" `stop` in the list.

```
>>> for elem in range(1, 5, 2):
...     print elem,
...
1 3
>>> range(1, 5, 2)
[1, 3]
```

*Pretty Print.* To print a list `a`, `print a` can be used, but the `pprint` and `scitools.pprint2` modules and their `pprint` function give a nicer layout of the output for long and nested lists. The `scitools.pprint2` module has the possibility to better control the output of floating-point numbers.

*If Tests.* The `if-elif-else` tests are used to "branch" the flow of statements. That is, different sets of statements are executed depending on whether a set of conditions is true or not.

```
def f(x):
    if x < 0:
        value = -1
    elif x >= 0 and x <= 1:
        value = x
    else:
        value = 1
    return value
```

*User-Defined Functions.* Functions are useful (i) when a set of commands are to be executed several times, or (ii) to partition the program into smaller pieces to gain better overview. Function arguments are local variables inside the function whose values are set when calling the function. Remember that when you write the function, the values of the arguments are not known. Here is an example of a function for polynomials of 2nd degree:

```
# function definition:
def quadratic_polynomial(x, a, b, c)
    value = a*x*x + b*x + c
    derivative = 2*a*x + b
    return value, derivative

# function call:
x = 1
p, dp = quadratic_polynomial(x, 2, 0.5, 1)
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)
```

The sequence of the arguments is important, unless all arguments are given as `name=value`.

Functions may have no arguments and/or no return value(s):

```
def print_date():
    """Print the current date in the format 'Jan 07, 2007'."""
    import time
    print time.strftime("%b %d, %Y")

# call:
print_date()
```

A common error is to forget the parentheses: `print_date` is the function object itself, while `print_date()` is a call to the function.

*Keyword Arguments.* Function arguments with default values are called keyword arguments, and they help to document the meaning of arguments in function calls. They also make it possible to specify just a subset of the arguments in function calls.

```
from math import exp, sin, pi

def f(x, A=1, a=1, w=pi):
    return A*exp(-a*x)*sin(w*x)

f1 = f(0)
x2 = 0.1
f2 = f(x2, w=2*pi)
```

```
f3 = f(x2, w=4*pi, A=10, a=0.1)
f4 = f(w=4*pi, A=10, a=0.1, x=x2)
```

The sequence of the keyword arguments can be arbitrary, and the keyword arguments that are not listed in the call get their default values according to the function definition. The “non-keyword arguments” are called positional arguments, which is `x` in this example. Positional arguments must be listed before the keyword arguments. However, also a positional argument can appear as `name=value` in the call (see the last line above), and this syntax allows any positional argument to be listed anywhere in the call.

*Terminology.* The important computer science terms in this chapter are

- list,
- tuple,
- nested list (and nested tuple),
- sublist (subtuple) or slice,
- `while` loop,
- `for` loop,
- list comprehension,
- boolean expression,
- function,
- method,
- return statement,
- positional arguments,
- keyword arguments,
- local and global variables,
- doc strings,
- `if` tests (branching),
- the `None` object.

### 2.4.2 Summarizing Example: Tabulate a Function

*Problem.* Make a program for evaluating the formula (1.1) for time points equally spaced by $\Delta t$ as long as $y \geq 0$. These time points and their associated $y$ values are to be stored in a nested list, to be printed out on the screen. Also search through the list to find the maximum $y(t)$ value.

*Solution.* We first present the program solving the problem stated above, and then we explain in detail how the program works. The code is found in the file `ball_table.py`.

```
g = 9.81;  v0 = 5
dt = 0.25

def y(t):
    return v0*t - 0.5*g*t**2

def table():
    data = []  # store [t, y] pairs in a nested list
    t = 0
    while y(t) >= 0:
        data.append([t, y(t)])
        t += dt
    return data

data = table()

for t, y in data:
    print '%5.1f  %5.1f' % (t, y)

# extract all y values from data:
y = [y for t, y in data]
print y
# find maximum y value:
ymax = 0
for yi in y:
    if yi > ymax:
        ymax = yi
print 'max y(t) =', ymax

data = table()  # this does not work now - why?
```

Recall that a program is executed from top to bottom, line by line, but the program flow may "jump around" because of functions and loops. Understanding the program flow in detail is a necessary and important ability if (when!) you need to find errors in a program that produces wrong results.

The present program starts with executing the first two lines, which bring the variables `g`, `v0`, and `dt` into play. Thereafter, two functions `y` and `table` are defined, but nothing inside these functions is computed. The computations in the function bodies are performed when we call the functions.

The first function call, `data = table()`, causes the program flow to jump into the `table` function and execute the statements in this function. When entering the `while` loop, the boolean expression[12] `y(t) >= 0` is evaluated. This expression requires the evaluation of `y(t)`, which causes the program flow to jump to the `y` function. Inside this function, the argument `t` is a local variable that has the value `0`, because the local variable `t` in the `table` function has the value `0` in the first call `y(t)`. The expression in the `return` statement in the `y` function is then evaluated to `0` and returned.

[12] Some experienced programmers may criticize the `table` function for having an unnecessary extra call to `y(t)` in each pass in the loop. Exercise 2.53 asks you to rewrite the function such that there is only one call to `y(t)` in the loop. However, in this summary section we have chosen to write code that is as easy to understand as possible, instead of writing as computationally efficient code as possible. We believe this strategy is beneficial for newcomers to programming.

We are now back to the boolean condition in the `while` loop. Since `y(t)` was evaluated to `0`, the condition reads `0 >= 0`, which is `True`. Therefore we are to execute the block of statements inside the loop. This block of statements is executed repeatedly until the loop condition is `False`. In each pass of the loop, the condition `y(t) >= 0` must be evaluated, and this task causes a call to the `y` function and an associated jump of the program flow. For some `t` value, `y(t)` will be negative, i.e., the loop condition is `False` and the program flow jumps to the first statement after the loop. This statement is `return data` in the present case.

The value of the call `table()` is now computed to be the object with name `data` inside the `table` function. We assign this object to a new global variable `data` in the statement `data = table()`. That is, there are two `data` variables in this program, one local in the `table` function and one global. The local `data` variable is demolished (along with the other local variable, `t`) when the program flow leaves the `table` function. Although the local `data` variable in the `table` function dies, the object that it refers to survives, because we "send out" this object from the function and bind it to a new, global name `data` in the main program. As long as we have some name that refers to an object, the object is alive and can be used. More aspects of this subject are discussed below.

The program flow proceeds to the `for` loop, where we run through all the pairs of `t, y` values in the nested list `data`. Recall that all elements of the `data` list are lists with two numbers, `t` and `y`.

The next statement applies a list comprehension to make a new list, `y`, holding all the `y` values that are in `data`. Again, we pass through all pairs `y, t` (i.e., elements) in `data`, but we place only the `y` part in the new list.

The next step is to find the maximum `y` value. We first set the maximum value `ymax` to the smallest relevant value, here it is 0 since $y \geq 0$. The `for` loop runs through all elements, and if an element is larger than `ymax`, `ymax` refers to this new element. When all elements in `y` are examined, `ymax` holds the largest value, and we can print it out. Since finding the largest (or smallest) element in a list is a frequently encountered task, Python has a special function `max` (or `min`) such that we could have written `ymax = max(y)` and hence avoided the `for` loop with the `if` test.

The final statement, `data = table()`, causes the program to abort with an error message

```
  File "ball_table.py", line 10, in table
    while y(t) >= 0:
TypeError: 'list' object is not callable
```

What has happended? The `table` function worked fine the first time we called it! As the error message tells us, the problem lies in the `while` loop line. You need some programming experience to understand what such an error message means and what the problem can be.

The global name `y` is originally used for a function in the program. However, after the call to `table`, we use the name `y` for several types of objects. First, `y` is used in the `for` loop and will there hold `float` objects. In the list comprehension, `y` is used in an identical way inside a `for` loop, and then `y` is the name for the resulting list! At the second call to `table`, the global name `y` refers to a list. The first time we use the global `y` inside the `table` function is in the loop condition `y(t) >= 0`. Here we try to call a function `y`, but `y` is a list, and the syntax of a call `y(t)` is illegal if `y` is a list. That is why the error message states that `'list' object is not callable`. Figure 2.3 illustrates the state of variables in the program after the first `table()` call and at the end.

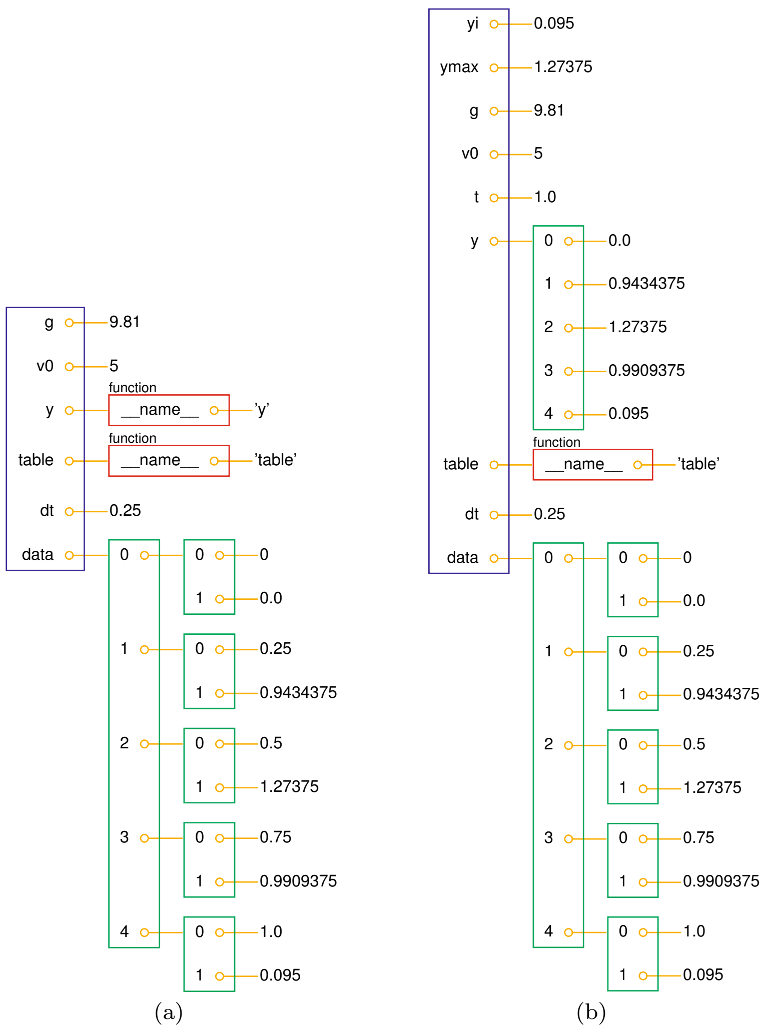



**Fig. 2.3** State of variables in `ball_table.py` (a) right after `table()` is called; (b) at the end. Note that `y` refers to a function in (a) and a list in (b).

How can we recover from this error? The simplest remedy is to give the `y` function another name, e.g., `yfunc`. Doing so, the program works. We still use the variable `y` for `float` and `list` objects, but this is okay as long as no errors arise.

In programming languages such as Fortran, C, C++, Java, and C#, any variable `y` is declared with a fixed type, and the problem that `y` is a function and later a number or list cannot occur. This principle removes some errors, but the flexibility of using the symbol `y` for different object types, whose conceptual name is conveniently taken as "y", makes the program easier to read and understand. (There is a Python package "Traits" that offers type checking of variables – in a more flexible way than what is possible with compile-time type checking as in Fortran, C, C++, Java, and C#.)

To better understand the flow of statements in a program, it can be handy to use a debugger. Appendix D.1 explains in detail how we can investigate the program flow in the present example with the aid of Python's built-in debugger.

### 2.4.3 How to Find More Python Information

This book contains only fragments of the Python language. When doing your own projects or exercises you will certainly feel the need for looking up more detailed information on modules, objects, etc. Fortunately, there is a lot of excellent documentation on the Python programming language. The primary reference is the official Python documentation website: `docs.python.org`. Here you can find a Python tutorial, the very useful *Python Library Reference*, an index of all modules that come with the basic Python distribution, and a Language Reference, to mention some. You should in particular discover the index of the Python Library Reference. When you wonder what functions you can find in a module, say the `math` module, you should go to this index, find the "math" keyword, and press the link. This brings you right to the official documentation of the `math` module. Similarly, if you want to look up more details of the printf formatting syntax, go to the index and follow the "printf-style formatting" index. A word of caution is probably necessary here: Reference manuals, such as the Python Library Reference, are very technical and written primarily for experts, so it can be quite difficult for a newbie to understand the information. An important ability is to browse such manuals and grab out the key information you are looking for, without being annoyed by all the text you do not understand. As with programming, reading manuals efficiently requires a lot of training.

A tool somewhat similar to the Python Library Reference is the `pydoc` program. In a terminal window you write

Terminal
```
Unix/DOS> pydoc math
```

In Python there are two possibilities, either[13]

```
In [1]: !pydoc math
```

or

```
In [2]: import math
In [3]: help(math)
```

The documentation of the complete `math` module is shown as plain text. If a specific function is wanted, we can ask for that directly, e.g., `pydoc math.tan`. Since `pydoc` is very fast, many prefer `pydoc` over webpages, but `pydoc` has often less information compared to the Python Library Reference.

There are also numerous books about Python. Beazley [1] is an excellent reference that improves and extends the information in the Python Library Reference. The "Learning Python" book [8] has been very popular for many years as an introduction to the language. There is a special webpage `http://wiki.python.org/moin/PythonBooks` listing most Python books on the market. A comprehensive book on the use of Python for doing scientific research is [5].

Quick references, which list "all" Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet: `http://rgruet.free.fr/PQR25/PQR2.5.html`.

The website `http://www.python.org/doc/` contains a list of useful Python introductions and reference manuals.

## 2.5 Exercises

**Exercise 2.1.** *Make a Fahrenheit–Celsius conversion table.*

Modify the `c2f_table_while.py` program so that it prints out a table with Fahrenheit degrees $0, 10, 20, \ldots, 100$ in the first column and the corresponding Celsius degrees in the second column. Name of program file: `c2f_table_while.py`. ◇

**Exercise 2.2.** *Generate odd numbers.*

Write a program that generates all odd numbers from 1 to `n`. Set `n` in the beginning of the program and use a `while` loop to compute the numbers. (Make sure that if `n` is an even number, the largest generated odd number is `n-1`.) Name of program file: `odd.py`. ◇

[13] Any command you can run in the terminal window can also be run inside IPython if you start the command with an exclamation mark.

**Exercise 2.3.** *Store odd numbers in a list.*

Modify the program from Exercise 2.2 to store the generated odd numbers in a list. Start with an empty list and use a `while` loop where you in each pass of the loop append a new element to the list. Finally, print the list elements to the screen. Name of program file: `odd_list1.py`. ◇

**Exercise 2.4.** *Generate odd numbers by the range function.*

Solve Exercise 2.3 by calling the `range` function to generate a list of odd numbers. Name of program file: `odd_list2.py`. ◇

**Exercise 2.5.** *Simulate operations on lists by hand.*

You are given the following program:

```
a = [1, 3, 5, 7, 11]
b = [13, 17]
c = a + b
print c
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d
```

Go through each statement and explain what is printed by the program. ◇

**Exercise 2.6.** *Make a table of values from formula (1.1).*

Write a program that prints a table of $t$ and $y(t)$ values from the formula (1.1) to the screen. Use 11 uniformly spaced $t$ values throughout the interval $[0, 2v_0/g]$, and fix the value of $v_0$. Name of program file: `ball_table1.py`. ◇

**Exercise 2.7.** *Store values from formula (1.1) in lists.*

In a program, make a list `t` with 6 $t$ values $0.1, 0.2, \ldots, 0.6$. Compute a corresponding list `y` of $y(t)$ values using formula (1.1). Write out a nicely formatted table of $t$ and $y$ values. Name of program file: `ball_table2.py`. ◇

**Exercise 2.8.** *Work with a list.*

Set a variable `primes` to a list containing the numbers 1, 3, 5, 7, 11, and 13. Write out each list element in a `for` loop. Assign 17 to a variable `p` and add `p` to the end of the list. Print out the whole new list. Name of program file: `primes.py`. ◇

**Exercise 2.9.** *Generate equally spaced coordinates.*

We want to generate $x$ coordinates between 1 and 2 with spacing 0.01. The $i$-th coordinate, $x_i$, is then $1 + ih$ where $h = 0.01$ and $i$ runs over integers $0, 1, \ldots, 100$. Compute the $x_i$ values and store them in a list. Hint: Use a `for` loop, and append each new $x_i$ value to a list, which is empty initially. Name of program file: `coor1.py`. ◇

**Exercise 2.10.** *Use a list comprehension to solve Exer. 2.9.*

The problem is the same as in Exercise 2.9, but now we want the $x_i$ values to be stored in a list using a list comprehension construct (see Chapter 2.1.6). Name of program file: `coor2.py`. ◇

**Exercise 2.11.** *Store data from Exer. 2.7 in a nested list.*

After having computed the two lists of $t$ and $y$ values in the program from Exercise 2.7, store the two lists in a new list `t1`. Write out a table of $t$ and $y$ values by traversing the data in the `t1` list. Thereafter, make a list `t2` which holds each row in the table of $t$ and $y$ values (`t1` is a list of table columns while `t2` is a list of table rows, as explained in Chapter 2.1.7). Write out the table by traversing the `t2` list. Name of program file: `ball_table3.py`. ◇

**Exercise 2.12.** *Compute a mathematical sum.*

The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{k}$:

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

This program does not work correctly. What are the three errors? (If you try to run the program, nothing will happen on the screen. Type Ctrl-C, i.e., hold down the Control (`Ctrl`) key and then type the `c` key, to stop a program.) Write a correct program. Name of program file: `compute_sum_while.py`.

There are two basic ways to find errors in a program: (i) read the program carefully and think about the consequences of each statement, and (ii) print out intermediate results and compare with hand calculations. First, try method (i) and find as many errors as you can. Then, try method (ii) for $M = 3$ and compare the evolution of `s` with your own hand calculations. ◇

**Exercise 2.13.** *Simulate a program by hand.*

Consider the following program for computing with interest rates:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

(a) Explain with words what type of mathematical problem that is solved by this program. Compare this computerized solution with the technique your high school math teacher would prefer.

(b) Use a pocket calculator (or use an interactive Python shell as substitute) and work through the program by hand. Write down the value of `amount` and `years` in each pass of the loop.

(c) Change the value of `p` to 5. Why will the loop now run forever? (See Exercise 2.12 for how to stop the program if you try to run it.) Make the program more robust against such errors.
(d) Make use of the operator `+=` wherever possible in the program.

Insert the text for the answers to (a) and (b) in a multi-line string in the program file. Name of program file: `interest_rate_loop.py`. ◇

**Exercise 2.14.** *Use a for loop in Exer. 2.12.*
Rewrite the corrected version of the program in Exercise 2.12 using a `for` loop over `k` values is used instead of a `while` loop. Name of program file: `compute_sum_for.py`. ◇

**Exercise 2.15.** *Index a nested lists.*
We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`. Name of program file: `index_nested_list.py`. ◇

**Exercise 2.16.** *Construct a double for loop over a nested list.*
Consider the list from Exercise 2.15. We can visit all elements of `q` using this nested `for` loop:

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`? Name of program file: `nested_list_iter.py`. ◇

**Exercise 2.17.** *Compute the area of an arbitrary triangle.*
An arbitrary triangle can be described by the coordinates of its three vertices: $(x_1, y_1)$, $(x_2, y_2)$, $(x_3, y_3)$. The area of the triangle is given by the formula

$$A = \frac{1}{2}\left[x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\right]. \tag{2.6}$$

Write a function `area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a nested list of the vertex coordinates. For example, `vertices` can be `[[0,0], [1,0], [0,2]]` if the three corners of the triangle have coordinates $(0,0)$, $(1,0)$, and $(0,2)$. Test the `area` function on a triangle with known area. Name of program file: `area_triangle.py`. ◇

**Exercise 2.18.** *Compute the length of a path.*
Some object is moving along a path in the plane. At $n$ points of time we have recorded the corresponding $(x, y)$ positions of the object:

$(x_0, y_0)$, $(x_1, y_2)$, …, $(x_{n-1}, y_{n-1})$. The total length $L$ of the path from $(x_0, y_0)$ to $(x_{n-1}, y_{n-1})$ is the sum of all the individual line segments ($(x_{i-1}, y_{i-1})$ to $(x_i, y_i)$, $i = 1, \ldots, n-1$):

$$L = \sum_{i=1}^{n-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}\,. \qquad (2.7)$$

Make a function `pathlength(x, y)` for computing $L$ according to the formula. The arguments `x` and `y` hold all the $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$ coordinates, respectively. Test the function on a triangular path with the four points $(1, 1)$, $(2, 1)$, $(1, 2)$, and $(1, 1)$. Name of program file: `pathlength.py`. ◇

**Exercise 2.19.** *Approximate pi.*

The value of $\pi$ equals the circumference of a circle with radius 1/2. Suppose we approximate the circumference by a polygon through $N+1$ points on the circle. The length of this polygon can be found using the `pathlength` function from Exercise 2.18. Compute $N+1$ points $(x_i, y_i)$ along a circle with radius 1/2 according to the formulas

$$x_i = \frac{1}{2}\cos(2\pi i/N), \quad y_i = \frac{1}{2}\sin(2\pi i/N), \quad i = 0, \ldots, N\,.$$

Call the `pathlength` function and write out the error in the approximation of $\pi$ for $N = 2^k$, $k = 2, 3, \ldots, 10$. Name of program file: `pi_approx.py`. ◇

**Exercise 2.20.** *Write a Fahrenheit-Celsius conversion table.*

Given a temperature $F$ in Fahrenheit degrees, the corresponding degrees in Celsius are found by solving (1.2) (on page 18) with respect to $C$, yielding formula (2.9). Many people use an approximate formula for quickly calculating the Celsius degrees: subtract 30 from the Fahrenheit degrees and divide by two, i.e.,

$$C = (F - 30)/2 \qquad (2.8)$$

We want to produce a table that compares the exact formula (2.9) and the rough approximation (2.8) for Fahrenheit degrees between 0 and 100 (in steps of, e.g., 10). Write a program for storing the $F$ values, the exact $C$ values, and the approximate $C$ values in a nested list. Print out a nicely formatted table by traversing the nested list with a `for` loop. Name of program file: `f2c_shortcut_table.py`. ◇

**Exercise 2.21.** *Convert nested list comprehensions to nested standard loops.*

Rewrite the generation of the nested list `q`,

```
q = [r**2 for r in [10**i for i in range(5)]]
```

by using standard `for` loops instead of list comprehensions. Name of program file: `listcomp2for.py`. ◇

**Exercise 2.22.** *Write a Fahrenheit–Celsius conversion function.*
The formula for converting Fahrenheit degrees to Celsius reads

$$C = \frac{5}{9}(F - 32)\,. \tag{2.9}$$

Write a function `C(F)` that implements this formula. To verify the implementation of `C(F)`, you can convert a Celsius temperature to Fahrenheit and then back to Celsius again using the `F(C)` function from Chapter 2.2.1 and the `C(F)` function implementing (2.9). That is, you can check that a temperature `c` equals `C(F(c))` (be careful with comparing real numbers with `==`, see Exercise 2.51). Name of program file: `c2f2c.py`. ◇

**Exercise 2.23.** *Write some simple functions.*
Write three functions:

1. `hw1`, which takes no arguments and returns the string `'Hello, World!'`
2. `hw2`, which takes no arguments and returns nothing, but the string `'Hello, World!'` is printed in the terminal window
3. `hw3`, which takes two string arguments and prints these two arguments separated by a comma

Use the following main program to test the three functions:

```
print hw1()
hw2()
hw3('Hello ', 'World!')
```

Name of program: `hw_func.py`. ◇

**Exercise 2.24.** *Write the program in Exer. 2.12 as a function.*
Define a Python function `s(M)` that computes the sum $s$ as defined in Exercise 2.12. Name of program: `compute_sum_func.py`. ◇

**Exercise 2.25.** *Implement a Gaussian function.*
Make a Python function `gauss(x, m=0, s=1)` for computing the Gaussian function (1.6) on page 45. Call `gauss` and print out the result for `x` equal to $-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5$, using default values for `m` and `s`. Name of program file: `Gaussian_function2.py`. ◇

**Exercise 2.26.** *Find the max and min values of a function.*
Write a function `maxmin(f, a, b, n=1000)` that returns the maximum and minimum values of a mathematical function `f(x)` (evaluated at `n` points) in the interval between `a` and `b`. The following test program

```
from math import cos, pi
print maxmin(cos, -pi/2, 2*pi)
```

should write out `(1.0, -1.0)`.

The `maxmin` function can compute a set of `n` coordinates between `a` and `b` stored in a list `x`, then compute `f` at the points in `x` and store the values in another list `y`. The Python functions `max(y)` and `min(y)` return the maximum and minimum values in the list `y`, respectively. Name of program file: `func_maxmin.py`. ◇

**Exercise 2.27.** *Explore the Python Library Reference.*

Suppose you want to make a program for printing out $\sin^{-1} x$ for $n$ $x$ values between 0 and 1. The `math` module has a function for computing $\sin^{-1} x$, but what is the right name of this function? Read Chapter 2.4.3 and use the `math` entry in the index of the Python Library Reference to find out how to compute $\sin^{-1} x$. Name of program file: `inverse_sine.py`. ◇

**Exercise 2.28.** *Make a function of the formula in Exer. 1.12.*

Implement the formula (1.8) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters $\rho$, $K$, $c$, and $T_w$ can be set as local (constant) variables inside the function. Let $t$ be returned from the function. Compute $t$ for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T = 25$ C). Name of program file: `egg_func.py`. ◇

**Exercise 2.29.** *Write a function for numerical differentiation.*

The formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.10}$$

can be used to find an approximate derivative of a mathematical function $f(x)$ if $h$ is small. Write a function `diff(f, x, h=1E-6)` that returns the approximation (2.10) of the derivative of a mathematical function represented by a Python function `f(x)`.

Apply (2.10) to differentiate $f(x) = e^x$ at $x = 0$, $f(x) = e^{-2x^2}$ at $x = 0$, $\cos x$ at $x = 2\pi$, and $f(x) = \ln x$ at $x = 1$. Use $h = 0.01$. In each case, write out the error, i.e., the difference between the exact derivative and the result of (2.10). Name of program file: `diff_f.py`. ◇

**Exercise 2.30.** *Write a function for numerical integration.*

An approximation to the integral of a function $f(x)$ over an interval $[a, b]$ can found by first approximating $f(x)$ by the straight line that goes through the end points $(a, f(a))$ and $(b, f(b))$, and then finding the area under the straight line (which is the area of a trapezoid). The resulting formula becomes

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a) + f(b))\,. \tag{2.11}$$

Write a function `integrate(f, a, b)` that returns this approximation to the integral. The argument `f` is a Python implementation `f(x)` of the mathematical function $f(x)$.

Compute the error, i.e., the difference between the approxmation (2.11) and the exact result, for Using (2.11), compute the following integrals $\int_0^{\ln 3} e^x dx$, $\int_0^{\pi} \cos x\, dx$, $\int_0^{\pi} \sin x\, dx$, and $\int_0^{\pi/2} \sin x\, dx$, In each case, write out the error, i.e., the difference between the exact integral and the approximation (2.11). Make rough sketches of (2.11) for each integral in order to understand how the method behaves in the different cases. Name of program file: `int_f.py`. ◇

**Exercise 2.31.** *Improve the formula in Exer. 2.30.*

We can easily improve the formula 2.11 from Exercise 2.30 by approximating the function $f(x)$ by a straight line from $(a, f(a))$ to the midpoint $(c, f(c))$ between $a$ and $b$, and then from the midpoint to $(b, f(b))$. The midpoint $c$ equals $\frac{1}{2}(a + b)$. The area under the two straight lines equals the area of two trapezoids. Derive a formula for this area and implement the formula in a function `integrate2(f, a, b)`. Run the examples from Exercise 2.30 and see how much better the new formula is. Name of program file: `int2_f.py`. ◇

**Exercise 2.32.** *Compute a polynomial via a product.*

Given $n$ roots $r_0, r_1, \ldots, r_n$ of a polynomial of degree $n$, the polynomial $p(x)$ can be computed by

$$p(x) = \prod_{i=0}^{n}(x - r_i) = (x - r_0)(x - r_1)\cdots(x - r_{n-1})(x - r_n)\,. \quad (2.12)$$

Store the roots $r_0, \ldots, r_n$ in a list and make a loop that computes the product in (2.12). Test the program on a polynomial with roots $-1$, 1, and 2. Name of program file: `polyprod.py`. ◇

**Exercise 2.33.** *Implement the factorial function.*

The factorial of $n$, written as $n!$, is defined as

$$n! = n(n-1)(n-2)\cdots 2\cdot 1, \quad (2.13)$$

with the special cases

$$1! = 1, \quad 0! = 1\,. \quad (2.14)$$

For example, $4! = 4\cdot 3\cdot 2\cdot 1 = 24$, and $2! = 2\cdot 1 = 2$. Write a function `fact(n)` that returns $n!$. Return 1 immediately if $x$ is 1 or 0, otherwise use a loop to compute $n!$. Name of program file: `fact.py`.

*Remark.* You can import a ready-made factorial function by

```
from scitools.std import factorial
```

This `factorial` function offers many different implementations, with different computational efficiency, for computing $x!$ (see the source code of the function for details). ◇

**Exercise 2.34.** *Compute velocity and acceleration from position data; one dimension.*

Let $x(t)$ be the position of an object moving along the $x$ axis. The velocity $v(t)$ and acceleration $a(t)$ can be approximately computed by the formulas

$$v(t) \approx \frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t}, \quad a(t) \approx \frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}, \tag{2.15}$$

where $\Delta t$ is a small time interval. As $\Delta \rightarrow 0$, the above formulas approach the first and second derivative of $x(t)$, which coincides with the well-known definitions of velocity and acceleration.

Write a function `kinematics(x, t, dt=1E-6)` for computing $x$, $v$, and $a$ time `t`, using the above formulas for $v$ and $a$ with $\Delta t$ corresponding to `dt`. Let the function return $x$, $v$, and $a$. Test the function with the position function $x(t) = e^{-(t-4)^2}$ and the time point $t = 5$ (use $\Delta t = 10^{-5}$). Name of program: `kinematics1.py`. ◇

**Exercise 2.35.** *Compute velocity and acceleration from position data; two dimensions.*

An object moves a long a path in the $xy$ plane such that at time $t$ the object is located at the point $(x(t), y(t))$. The velocity vector in the plane, at time $t$, can be approximated as

$$v(t) \approx \left(\frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t}, \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}\right). \tag{2.16}$$

The acceleration vector in the plane, at time $t$, can be approximated as

$$a(t) \approx \left(\frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}, \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2}\right). \tag{2.17}$$

Here, $\Delta t$ is a small time interval.

Make a function `kinematics(x, y, t, dt=1E-6)` for computing the velocity and acceleration of the object according to the formulas above (`t` corresponds to $t$, and `dt` corresponds to $\Delta t$). The function should return three 2-tuples holding the position, the velocity, and the acceleration, all at time `t`. Test the function for the motion along a circle with radius $R$ and absolute velocity $R\omega$: $x(t) = R\cos\omega t$ and $y(t) = R\sin\omega t$. Compute the velocity and acceleration for $t = 0$ and $t = \pi$ using $R = 1$ and $\omega = 2\pi$. Name of program: `kinematics2.py`. ◇

**Exercise 2.36.** *Express a step function as a Python function.*

The following "step" function is known as the Heaviside function and is widely used in mathematics:

$$H(x) = \begin{cases} 0, \; x < 0 \\ 1, \; x \geq 0 \end{cases} \tag{2.18}$$

Write a Python function `H(x)` that evaluates the formula for $H(x)$. Test your implementation for $x = -\frac{1}{2}, 0, 10$. Name of program file: `Heaviside.py`. ◇

**Exercise 2.37.** *Rewrite a mathematical function.*

We consider the $L(x;n)$ sum as defined in Chapter 2.2.4 and the corresponding function `L2(x, epsilon)` function from Chapter 2.2.6. The sum $L(x;n)$ can be written as

$$L(x;n) = \sum_{i=1}^{n} c_i, \quad c_i = \frac{1}{i}\left(\frac{x}{1+x}\right)^i .$$

Derive a relation between $c_i$ and $c_{i-1}$,

$$c_i = ac_{i-1},$$

where $a$ is an expression involving $i$ and $x$. This relation between $c_i$ and $c_{i-1}$ means that we can start with `term` as $c_1$, and then in each pass of the loop implementing the sum $\sum_i c_i$ we can compute the next term $c_i$ in the sum as

```
term = a*term
```

Rewrite the `L2` function to make use of this alternative computation. Compare the new version with the original one to verify the implementation. Name of program file: `L2_recursive.py`. ◇

**Exercise 2.38.** *Make a table for approximations of* $\cos x$.

The function $\cos(x)$ can be approximated by the sum

$$C(x;n) = \sum_{j=0}^{n} c_j, \tag{2.19}$$

where

$$c_j = -c_{j-1}\frac{x^2}{2j(2j-1)}, \quad j = 1, 2, \ldots, n,$$

and $c_0 = 0$. Make a Python function for computing $C(x;n)$. (Hint: Represent $c_j$ by a variable `term`, make updates `term = -term*...` inside a `for` loop, and accumulate the `term` variable in a variable for the sum.)

Also make a function for writing out a table of the errors in the approximation $C(x;n)$ of $\cos(x)$ for some $x$ and $n$ values given as arguments to the function. Let the $x$ values run downward in the rows

and the $n$ values to the right in the columns. For example, a table for $x = 4\pi, 6\pi, 8\pi, 10\pi$ and $n = 5, 25, 50, 100, 200$ can look like

```
     x           5           25          50          100         200
 12.5664    1.61e+04    1.87e-11    1.74e-12    1.74e-12    1.74e-12
 18.8496    1.22e+06    2.28e-02    7.12e-11    7.12e-11    7.12e-11
 25.1327    2.41e+07    6.58e+04   -4.87e-07   -4.87e-07   -4.87e-07
 31.4159    2.36e+08    6.52e+09    1.65e-04    1.65e-04    1.65e-04
```

Observe how the error increases with $x$ and decreases with $n$. Name of program file: `cossum.py`. ◇

**Exercise 2.39.** *Implement Exer. 1.13 with a loop.*

Make a function for evaluating $S(t; n)$ as defined in Exercise 1.13 on page 46, using a loop to sum up the $n$ terms. The function should take $t$, $T$, and $n$ as arguments and return $S(t; n)$ and the error in the approximation of $f(t)$. Make a main program that sets $T = 2$ and writes out a table of the approximation errors for $n = 1, 5, 20, 50, 100, 200, 500, 1000$ and $t = 1.01, 1.1, 1.8$. Use a row for each $n$ value and a column for each $t$ value. Name of program file: `compute_sum_S.py`. ◇

**Exercise 2.40.** *Determine the type of objects.*

Consider the following calls to the `makelist` function from page 76:

```
l1 = makelist(0, 100, 1)
l2 = makelist(0, 100, 1.0)
l3 = makelist(-1, 1, 0.1)
l4 = makelist(10, 20, 20)
l5 = makelist([1,2], [3,4], [5])
l6 = makelist((1,-1,1), ('myfile.dat', 'yourfile.dat'))
l7 = makelist('myfile.dat', 'yourfile.dat', 'herfile.dat')
```

Determine in each case what type of objects that become elements in the returned list and what the contents of `value` is after one pass in the loop.

Hint: Simulate the program by hand and check out in an interactive session what type of objects that result from the arithmetics. It is only necessary to simulate one pass of the loop to answer the questions. Some of the calls will lead to infinite loops if you really execute the `makelist` calls on a computer.

This exercise demonstrates that we can write a function and have in mind certain types of arguments, here typically `int` and `float` objects. However, the function can be used with other (originally unintended) arguments, such as lists and strings in the present case, leading to strange and irrelevant behavior (the problem here lies in the boolean expression `value <= stop` which is meaningless for some of the arguments).

◇

**Exercise 2.41.** *Implement the sum function.*

The standard Python function called `sum` takes a list as argument and computes the sum of the elements in the list:

```
>>> sum([1,3,5,-5])
4
```

Implement your own version of `sum`. Name of program: `sum.py`. ◇

**Exercise 2.42.** *Find the max/min elements in a list.*

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. The purpose of this exercise is to write your own `max` and `min` function. Use the following technique: Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, make `max_elem` refer to that element. Use a similar technique to compute the minimum element. Collect the two pieces of code in functions. Name of program file: `maxmin_list.py`. ◇

**Exercise 2.43.** *Demonstrate list functionality.*

Create an interactive session where you demonstrate the effect of each of the operations in Table 2.1 on page 92. Use IPython and log the results (see Exercise 1.11). Name of program file: `list_demo.py`. ◇

**Exercise 2.44.** *Write a sort function for a list of 4-tuples.*

Below is a list of the nearest stars and some of their properties. The list elements are 4-tuples containing the name of the star, the distance from the sun in light years, the apparent brightness, and the luminosity. The apparent brightness is how bright the stars look in our sky compared to the brightness of Sirius A. The luminosity, or the true brightness, is how bright the stars would look if all were at the same distance compared to the Sun. The list data are found in the file `stars.list`, which looks as follows:

```
data = [
('Alpha Centauri A',    4.3,  0.26,      1.56),
('Alpha Centauri B',    4.3,  0.077,     0.45),
('Alpha Centauri C',    4.2,  0.00001,   0.00006),
("Barnard's Star",      6.0,  0.00004,   0.0005),
('Wolf 359',            7.7,  0.000001,  0.00002),
('BD +36 degrees 2147', 8.2,  0.0003,    0.006),
('Luyten 726-8 A',      8.4,  0.000003,  0.00006),
('Luyten 726-8 B',      8.4,  0.000002,  0.00004),
('Sirius A',            8.6,  1.00,      23.6),
('Sirius B',            8.6,  0.001,     0.003),
('Ross 154',            9.4,  0.00002,   0.0005),
]
```

The purpose of this exercise is to sort this list with respect to distance, apparent brightness, and luminosity.

To sort a list `data`, one can call `sorted(data)`, which returns the sorted list (cf. Table 2.1). However, in the present case each element is a 4-tuple, and the default sorting of such 4-tuples result in a list with the stars appearing in alphabethic order. We need to sort with respect to the 2nd, 3rd, or 4th element of each 4-tuple. If a tailored

sort mechanism is necessary, we can provide our own sort function as a second argument to `sorted`, as in `sorted(data, mysort)`. Such a tailored sort function `mysort` must take two arguments, say `a` and `b`, and returns $-1$ if `a` should become before `b` in the sorted sequence, 1 if `b` should become before `a`, and 0 if they are equal. In the present case, `a` and `b` are 4-tuples, so we need to make the comparison between the right elements in `a` and `b`. For example, to sort with respect to luminosity we write

```
def mysort(a, b):
    if a[3] < b[3]:
        return -1
    elif a[3] > b[3]:
        return 1
    else:
        return 0
```

Write the complete program which initializes the data and writes out three sorted tables: star name versus distance, star name versus apparent brightness, and star name versus luminosity. Name of program file: `sorted_stars_data.py`. ◇

**Exercise 2.45.** *Find prime numbers.*

The *Sieve of Eratosthenes* is an algorithm for finding all prime numbers less than or equal to a number $N$. Read about this algorithm on Wikipedia and implement it in a Python program. Name of program file: `find_primes.py`. ◇

**Exercise 2.46.** *Condense the program in Exer. 2.14.*

The program in Exercise 2.14 can be greatly condensed by applying the `sum` function to a list of all the elements $1/k$ in the sum $\sum_{k=1}^{M} \frac{1}{k}$:

```
print sum([1.0/k for k in range(1, M+1, 1)])
```

The list comprehension here first builds a list of all elements in the sum, and this may consume a lot of memory in the computer. Python offers an alternative syntax

```
print sum(1.0/k for k in xrange(1, M+1, 1))
```

where we get rid of the list produced by a list comprehension. We also get rid of the list returned by `range`, because `xrange` generates a sequence of the same integers as `range`, but the integers are not stored in a list (they are generated as they are needed). For very large lists, `xrange` is therefore more efficient than `range`.

The purpose of this exercise is to compare the efficiency of the two calls to `sum` as listed above. Use the `time` module from Appendix E.6.1 to measure the CPU time spent by each construction. Write out $M$ and the CPU time for $M = 10^4, 10^6, 10^8$. (Your computer may become very busy and "hang" in the last case because the list comprehension and `range` calls demand $2M$ numbers to be stored, which

may exceed the computer's memory capacity.) Name of program file: `compute_sum_compact.py`. ◇

**Exercise 2.47.** *Values of boolean expressions.*
Explain the outcome of each of the following boolean expressions:

```
C = 41
C == 40
C != 40 and C < 41
C != 40 or  C < 41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Note: It makes sense to compare `True` and `False` to the integers 0 and 1, but not other integers (e.g., `True == 12` is `False` although the *integer* 12 evaluates to `True` in a boolean context, as in `bool(12)` or `if 12`). ◇

**Exercise 2.48.** *Explore round-off errors from a large number of inverse operations.*
Maybe you have tried to hit the square root key on a calculator multiple times and then squared the number again an equal number of times. These set of inverse mathematical operations should of course bring you back to the starting value for the computations, but this does not always happen. To avoid tedious pressing of calculator keys we can let a computer automate the process. Here is an appropriate program:

```
from math import sqrt
for n in range(60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print '%d times sqrt and **2: %.16f' % (n, r)
```

Explain with words what the program does. Then run the program. Round-off errors are here completely destroying the calculations when `n` is large enough! Investigate the case when we come back to 1 instead of 2 by fixing the `n` value and printing out `r` in both `for` loops over `i`. Can you now explain why we come back to 1 and not 2? Name of program file: `repeated_sqrt.py`. ◇

**Exercise 2.49.** *Explore what zero can be on a computer.*
Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print '...............', eps
    eps = eps/2.0
print 'final eps:', eps
```

Explain with words what the code is doing, line by line. Then examine the output. How can it be that the "equation" $1 \neq 1 + \text{eps}$ is not true? Or in other words, that a number of approximately size $10^{-16}$ (the final `eps` value when the loop terminates) gives the same result as if `eps`[14] were zero? Name of program file: `machine_zero.py`.

If somebody shows you this interactive session

```
>>> 0.5 + 1.45E-22
0.5
```

and claims that Python cannot add numbers correctly, what is your answer? ◇

**Exercise 2.50.** *Resolve a problem with a function.*

Consider the following interactive session:

```
>>> def f(x):
...     if 0 <= x <= 2:
...         return x**2
...     elif 2 < x <= 4:
...         return 4
...     elif x < 0:
...         return 0
...
>>> f(2)
4
>>> f(5)
>>> f(10)
```

Why do we not get any output when calling `f(5)` and `f(10)`? (Hint: Save the `f` value in a variable `r` and write `print r`.) ◇

**Exercise 2.51.** *Compare two real numbers on a computer.*

Consider the following simple program inspired by Chapter 1.4.3:

```
a = 1/947.0*947
b = 1
if a != b:
    print 'Wrong result!'
```

Try to run this example!

One should never compare two floating-point objects directly using `==` or `!=`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if $|a - b|$ is sufficiently small, i.e., if $a$ and $b$ are "close enough" to be considered equal. Modify the test according to this idea.

[14] This nonzero `eps` value is called *machine epsilon* or *machine zero* and is an important parameter to know, especially when certain mathematical techniques are applied to control round-off errors.

Thereafter, read the documentation of the function `float_eq` from SciTools: `scitools.numpyutils.float_eq` (see page 98 for how to bring up the documentation of a module or a function in a module). Use this function to check whether two real numbers are equal within a tolerance. Name of program file: `compare_float.py`. ◇

**Exercise 2.52.** *Use None in keyword arguments.*

Consider the functions `L(x, n)` and `L2(x, epsilon)` from Chapter 2.2.6, whose program code is found in the file `lnsum.py`. Let us make a more flexible function `L3` where we can either specify a tolerance `epsilon` or a number of terms `n` in the sum, and we can choose whether we want the sum to be returned or the sum and the number of terms. The latter set of return values is only meaningful with `epsilon` and not `n` is specified. The starting point for all this flexibility is to have some keyword arguments initialized to an "undefined" value that can be recognized:

```
def L3(x, n=None, epsilon=None, return_n=False):
```

You can test if `n` is given using the phrase[15]

```
    if n is not None:
```

A similar construction can be used for `epsilon`. Print error messages for incompatible settings when `n` *and* `epsilon` are `None` (none given) or `not None` (both given). Name of program file: `L3_flexible.py`. ◇

**Exercise 2.53.** *Improve the program from Ch. 2.4.2.*

The `table` function in the program from Chapter 2.4.2 evaluates `y(t)` twice for the same value of the argument `t`. This waste of work has no practical consequences in this little program because the `y` function is so fast to calculate. However, mathematical computations soon lead to programs that takes minutes, hours, days, and even weeks to run. In those cases one should avoid repeating calculations. It is in general considered a good habit to make programs efficient, at least to remove obvious redundant calculations.

Write a new `table` function that has only one `y(t)` in the `while` loop. (Hint: Store the `y(t)` value in a variable.)

How can you make the `y` function more efficient by reducing the number of arithmetic operations? (Hint: Factorize $t$ and precompute `0.5*g` in a global variable.) Name of program file: `ball_table_efficient.py`. ◇

**Exercise 2.54.** *Interpret a code.*

The function `time` in the module `time` returns the number of seconds since a particular date (called the Epoch, which is January 1,

[15] One can also apply `if n != None`, but the `is` operator is most common (it tests if `n` and `None` are identical objects, not just objects with equal contents).

1970 on many types of computers). Python programs can therefore use `time.time()` to mimic a stop watch. Another function, `time.sleep(n)` causes the program to "sleep" `n` seconds and is handy to insert a pause. Use this information to explain what the following code does:

```
import time
t0 = time.time()
while time.time() - t0 < 10:
    print '....I like while loops!'
    time.sleep(2)
print 'Oh, no - the loop is over.'
```

How many times is the `print` statement inside the loop executed? Now, copy the code segment and change the `<` sign in the loop condition to a `>` sign. Explain what happens now. Name of program: `time_while.py`.
◇

**Exercise 2.55.** *Explore problems with inaccurate indentation.*

Type in the following program in a file and check carefully that you have exactly the same spaces:

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
      print C, F
C = C + dC
```

Run the program. What is the first problem? Correct that error. What is the next problem? What is the cause of that problem? (See Exercise 2.12 for how to stop a hanging program.)

The lesson learned from this exercise is that one has to be very careful with indentation in Python programs! Other computer languages usually enclose blocks belonging to loops and `if`-tests in curly braces, parentheses, or BEGIN-END marks. Python's convention with using solely indentation contributes to visually attractive, easy-to-read code, at the cost of requiring a pedantic attitude to blanks from the programmer. ◇

**Exercise 2.56.** *Find an error in a program.*

Consider the following program for computing

$$f(x) = e^{rx}\sin(mx) + e^{sx}\sin(nx),$$

```
def f(x, m, n, r, s):
    return expsin(x, r, m) + expsin(x, s, n)

x = 2.5
print f(x, 0.1, 0.2, 1, 1)

from math import exp, sin

def expsin(x, p, q):
    return exp(p*x)*sin(q*x)
```

Running this code results in

```
NameError: global name 'expsin' is not defined
```

What is the problem? Simulate the program flow by hand or use the debugger to step from line to line. Correct the program. ◇

**Exercise 2.57.** *Find programming errors.*

What is wrong in the following code segments? Try first to find the errors in each case by visual inspection of the code. Thereafter, type in the code snippet and test it out in an interactive Python shell.

```
def f(x)
    return 1+x**2;
```

*Case 1:*

```
def f(x):
    term1 = 1
  term2 = x**2
  return term1 + term2
```

*Case 2:*

```
def f(x, a, b):
    return a + b*x

print f(1), f(2), f(3)
```

*Case 3:*

```
def f(x, w):
    from math import sin
    return sin(w*x)

f = 'f(x, w)'
w = 10
x = 0.1
print f(x, w)
```

*Case 4:*

```
from math import *

def log(message):
    print message

print 'The logarithm of 1 is', log(1)
```

*Case 5:*

```
import time

def print_CPU_time():
    print 'CPU time so far in the program:', time.clock()

print_CPU_time;
```

*Case 6:* ◇

**Exercise 2.58.** *Simulate nested loops by hand.*

Go through the code below by hand, statement by statement, and calculate the numbers that will be printed.

```
n = 3
for i in range(-1, n):
    if i != 0:
        print i

for i in range(1, 13, 2*n):
    for j in range(n):
        print i, j

for i in range(1, n+1):
    for j in range(i):
        if j:
            print i, j

for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                print i, j, k
```

You may use a debugger, see Appendix D.1, to step through the code and to see what happens. ◇

**Exercise 2.59.** *Explore punctuation in Python programs.*

Some of the following assignments work and some do not. Explain in each case why the assignment works/fails and, if it works, what kind of object `x` refers to and what the value is if we do a `print x`.

```
x = 1
x = 1.
x = 1;
x = 1!
x = 1?
x = 1:
x = 1,
```

Hint: Explore the statements in an interactive Python shell. ◇

**Exercise 2.60.** *Investigate a for loop over a changing list.*

Study the following interactive session and explain in detail what happens in each pass of the loop, and use this explanation to understand the output.

```
>>> numbers = range(10)
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for n in numbers:
...     i = len(numbers)/2
...     del numbers[i]
...     print 'n=%d, del %d' % (n,i), numbers
...
n=0, del 5 [0, 1, 2, 3, 4, 6, 7, 8, 9]
```

```
n=1, del 4 [0, 1, 2, 3, 6, 7, 8, 9]
n=2, del 4 [0, 1, 2, 3, 7, 8, 9]
n=3, del 3 [0, 1, 2, 7, 8, 9]
n=8, del 3 [0, 1, 2, 8, 9]
```

The message in this exercise is to *never modify a list that is used in a* `for` *loop*. Modification is indeed technically possible, as we show above, but you really need to know what you do – to avoid getting frustrated by strange program behavior. ◇

# Input Data and Error Handling

# 3

Recall our first program for evaluating the formula (1.2) on page 18 in Chapter 1:

```
C = 21
F = (9/5)*C + 32
print F
```

In this program, `C` is input data in the sense that `C` must be known before the program can perform the calculation of `F`. The results produced by the program, here `F`, constitute the output data.

Input data can be hardcoded in the program as we do above. That is, we explicitly set variables to specific values (`C = 21`). This programming style may be suitable for small programs. In general, however, it is considered good practice to let a user of the program provide input data when the program is running. There is then no need to modify the program itself when a new set of input data is to be explored[1].

This chapter starts with describing three different ways of reading data into a program: (i) letting the user answer questions in a dialog in the terminal window (Chapter 3.1), (ii) letting the user provide input on the command line (Chapter 3.2), and (iii) letting the user write input data in a graphical interface (Chapter 3.4). A fourth method is to read data from a file, but this topic is left for Chapter 6.

Even if your program works perfectly, wrong input data from the user may cause the program to produce wrong answers or even crash. Checking that the input data are correct is important, and Chapter 3.3 tells you how to do this with so-called exceptions.

The Python programming environment is organized as a big collection of modules. Organizing your own Python software in terms of

[1] Programmers know that any modification of the source code has a danger of introducing errors, so it is a good rule to change as little as possible in a program that works.

modules is therefore a natural and wise thing to do. Chapter 3.5 tells you how easy it is to make your own modules.

All the program examples from the present chapter are available in files in the `src/input` folder.

## 3.1 Asking Questions and Reading Answers

One of the simplest ways of getting data into a program is to ask the user a question, let the user type in an answer, and then read the text in that answer into a variable in the program. These tasks are done by calling a function with name `raw_input`. A simple example involving the temperature conversion program above will quickly show how to use this function.

### 3.1.1 Reading Keyboard Input

We may ask the user a question `C=?` and wait for the user to enter a number. The program can then read this number and store it in a variable `C`. These actions are performed by the statement

```
C = raw_input('C=? ')
```

The `raw_input` function always returns the user input as a string object. That is, the variable `C` above refers to a string object. If we want to compute with this `C`, we must convert the string to a floating-point number: `C = float(C)`. A complete program for reading `C` and computing the corresponding degrees in Fahrenheit now becomes

```
C = raw_input('C=? ')
C = float(C)
F = (9./5)*C + 32
print F
```

In general, the `raw_input` function takes a string as argument, displays this string in the terminal window, waits until the user presses the Return key, and then returns a string object containing the sequence of characters that the user typed in.

The program above is stored in a file called `c2f_qa.py` (the `qa` part of the name reflects "question and answer"). We can run this program in several ways, as described in Chapter 1.1.5 and Appendix E.1. The convention in this book is to indicate the execution by writing the program name only, but for a real execution you need to do more: write `run` before the program name in an interactive IPython session, or write `python` before the program name in a terminal session. Here is the execution of our sample program and the resulting dialog with the user:

Terminal

```
c2f_qa.py
C=? 21
69.8
```

In this particular example, the `raw_input` function reads the characters `21` from the keyboard and returns the string `'21'`, which we refer to by the variable `C`. Then we create a new `float` object by `float(C)` and let the name `C` refer to this `float` object, with value 21.

You should now try out Exercises 3.1, 3.4, and 3.6 to make sure you understand how `raw_input` behaves.

### 3.1.2 The Magic "eval" Function

Python has a function `eval`, which takes a string as argument and evaluates this string as a Python expression. This functionality can be used to turn input into running code on the fly. To realize what it means, we invoke an interactive session:

```
>>> r = eval('1+2')
>>> r
3
>>> type(r)
<type 'int'>
```

The result of `r = eval('1+2')` is the same as if we had written `r = 1+2` directly:

```
>>> r = 1+2
>>> r
3
>>> type(r)
<type 'int'>
```

In general, any valid Python expression stored as text in a string `s` can be turned into Python code by `eval(s)`. Here is an example where the string to be evaluated is `'2.5'`, which causes Python to see `r = 2.5` and make a `float` object:

```
>>> r = eval('2.5')
>>> r
2.5
>>> type(r)
<type 'float'>
```

If we put a string, enclosed in quotes, inside the expression string, the result is a string object:

```
>>>
>>> r = eval('"math programming"')
>>> r
'math programming'
>>> type(r)
<type 'str'>
```

Note that we must use two types of quotes: first double quotes to mark `math programming` as a string object and then another set of quotes, here single quotes (but we could also have used triple single quotes), to embed the text `"math programming"` inside a string. It does not matter if we have single or double quotes as inner or outer quotes, i.e., `'"..."'` is the same as `"'...'"`, because `'` and `"` are interchangeable as long as a pair of either type is used consistently.

Writing just

```
>>> r = eval('math programming')
```

is the same as writing

```
>>> r = math programming
```

which is an invalid expression. Python will in this case think that `math` and `programming` are two (undefined) variables, and setting two variables next to each other with a space in between is invalid Python syntax. However,

```
>>> r = 'math programming'
```

is valid syntax, as this is how we initialize a string `r` in Python. To repeat, if we put the valid syntax `'math programming'` inside a string,

```
s = "'math programming'"
```

`eval(s)` will evaluate the text inside the double quotes as `'math programming'`, which yields a string.

Let us proceed with some more examples. We can put the initialization of a list inside quotes and use `eval` to make a `list` object:

```
>>> r = eval('[1, 6, 7.5]')
>>> r
[1, 6, 7.5]
>>> type(r)
<type 'list'>
```

Again, the assignment to `r` is equivalent to writing

```
>>> r = [1, 6, 7.5]
```

We can also make a `tuple` object by using tuple syntax (standard parentheses instead of brackets):

```
>>> r = eval('(-1, 1)')
>>> r
(-1, 1)
>>> type(r)
<type 'tuple'>
```

Another example reads

```
>>> from math import sqrt
>>> r = eval('sqrt(2)')
>>> r
1.4142135623730951
>>> type(r)
<type 'float'>
```

At the time we run `eval('sqrt(2)')`, this is the same as if we had written

```
>>> r = sqrt(2)
```

directly, and this is valid syntax only if the `sqrt` function is defined. Therefore, the import of `sqrt` prior to running `eval` is important in this example.

So, why is the `eval` function so useful? Recall the `raw_input` function, which always returns a string object, which we often must explicitly transform to a different type, e.g., an `int` or a `float`. Sometimes we want to avoid specifying one particular type. The `eval` function can then be of help: we feed the returned string from `raw_input` to `eval` and let the latter function interpret the string and convert it to the right object. An example may clarify the point. Consider a small program where we read in two values and add them. The values could be strings, floats, integers, lists, and so forth, as long as we can apply a `+` operator to the values. Since we do not know if the user supplies a string, float, integer, or something else, we just convert the input by `eval`, which means that the user's syntax will determine the type. The program goes as follows (`add_input.py`):

```
i1 = eval(raw_input('Give input: '))
i2 = eval(raw_input('Give input: '))
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

Observe that we write out the two supplied values, together with the types of the values (obtained by `eval`), and the sum. Let us run the program with an integer and a real number as input:

Terminal

```
add_input.py
Give input: 4
Give input: 3.1
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 7.1
```

The string `'4'`, returned by the first call to `raw_input`, is interpreted as an `int` by `eval`, while `'3.1'` gives rise to a `float` object.

Supplying two lists also works fine:

Terminal

```
add_input.py
Give input: [-1, 3.2]
```

```
Give input: [9,-2,0,0]
<type 'list'> + <type 'list'> becomes <type 'list'>
with value [-1, 3.2000000000000002, 9, -2, 0, 0]
```

If we want to use the program to add two strings, the strings must be enclosed in quotes for `eval` to recognize the texts as string objects (without the quotes, `eval` aborts with an error):

Terminal

```
add_input.py
Give input: 'one string'
Give input: " and another string"
<type 'str'> + <type 'str'> becomes <type 'str'>
with value one string and another string
```

Not all objects are meaningful to add:

Terminal

```
add_input.py
Give input: 3.2
Give input: [-1,10]
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: unsupported operand type(s) for +: 'float' and 'list'
```

Another important example on the usefulness of `eval` is to turn formulas, given as input, into mathematics in the program. Consider the program

```
formula = raw_input('Give a formula involving x: ')
x = eval(raw_input('Give x: '))
from math import *   # make all math functions available
result = eval(formula)
print '%s for x=%g yields %g' % (formula, x, result)
```

First, we ask the reader to provide a formula, e.g., `2*sin(x)+1`. The result is a string object referred to by the `formula` variable. Then, we ask for an `x` value, typically a real number resulting in a `float` object. The key statement involves `eval(formula)`, which in the present example evaluates the expression `2*sin(x)+1`. The `x` variable is defined, and the `sin` function is also defined because of the `import` statement. Let us try to run the program:

Terminal

```
eval_formula.py
Give a formula involving x: 2*sin(x)+1
Give x: 3.14
2*sin(x)+1 for x=3.14 yields 1.00319
```

Another important application of `eval` occurs in Chapter 3.2.1.

### 3.1.3 The Magic "exec" Function

Having presented `eval` for turning strings into Python code, we take the opportunity to also describe the related `exec` function to execute a string containing arbitrary Python code, not only an expression. Suppose the user can write a formula as input to the program, and that we want to turn this formula into a callable Python function. That is, writing `sin(x)*cos(3*x) + x**2` as the formula, we would like to get a function

```
def f(x):
    return sin(x)*cos(3*x) + x**2
```

This is easy with `exec`:

```
formula = raw_input('Write a formula involving x: ')
code = """
def f(x):
    return %s
""" % formula
exec(code)
```

If we respond with the text `sin(x)*cos(3*x) + x**2` to the question, `formula` will hold this text, which is inserted into the `code` string such that it becomes

```
"""
def f(x):
    return sin(x)*cos(3*x) + x**2
"""
```

Thereafter, `exec(code)` executes the code as if we had written the contents of the `code` string directly into the program by hand. With this technique, we can turn any user-given formula into a Python function!

Let us try out such code generation on the fly. We add a `while` loop to the previous code snippet defining `f(x)` such that we can provide `x` values and get `f(x)` evaluated:

```
x = 0
while x is not None:
    x = eval(raw_input('Give x (None to quit): '))
    if x is not None:
        print 'f(%g)=%g' % (x, f(x))
```

As long as we provide numbers as input for `x`, we evaluate the `f(x)` function, but when we provide the text `None`, `x` becomes a `None` object and the test in the `while` loop fails, i.e., the loop terminates. The complete program is found in the file `user_formula.py`. Here is a sample run:

Terminal

```
user_formula.py
Write a formula involving x: x**4 + x
Give x (None to quit): 1
```

```
f(1)=2
Give x (None to quit): 4
f(4)=260
Give x (None to quit): 2
f(2)=18
Give x (None to quit): None
```

### 3.1.4 Turning String Expressions into Functions

The examples in the previous section indicate that it can be handy to ask the user for a formula and turn that formula into a Python function. Since this operation is so useful, we have made a special tool that hides the technicalities. The tool is named `StringFunction` and works as follows:

```
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)   # turn formula into function f(x)
```

The `f` object now behaves as an ordinary Python function of `x`:

```
>>> f(0)
0.0
>>> f(pi)
2.8338239229952166e-15
>>> f(log(1))
0.0
```

Expressions involving other independent variables than `x` are also possible. Here is an example with the function $g(t) = Ae^{-at}\sin(\omega x)$:

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
                   independent_variable='t',
                   A=1, a=0.1, omega=pi, x=0.5)
```

The first argument is the function formula, as before, but now we need to specify the name of the independent variable (`'x'` is default). The other parameters in the function ($A$, $a$, $\omega$, and $x$) must be specified with values, and we use keyword arguments, consistent with the names in the function formula, for this purpose. Any of the parameters `A`, `a`, `omega`, and `x` can be changed later by calls like

```
g.set_parameters(omega=0.1)
g.set_parameters(omega=0.1, A=5, x=0)
```

Calling `g(t)` works as if `g` were a plain Python function of `t`, which "remembers" all the parameters `A`, `a`, `omega`, and `x`, and their values. You can use `pydoc` (see page 98) to bring up more documentation on the possibilities with `StringFunction`. Just run

```
pydoc scitools.StringFunction.StringFunction
```

A final important point is that `StringFunction` objects are as computationally efficient as hand-written Python functions[2].

## 3.2 Reading from the Command Line

Programs running on Unix computers usually avoid asking the user questions. Instead, input data are very often fetched from the *command line*. This section explains how we can access information on the command line in Python programs.

### 3.2.1 Providing Input on the Command Line

We look at the Celsius-Fahrenheit conversion program again. The idea now is to provide the Celsius input temperature as a *command-line argument* right after the program name. That is, we write the program name, here `c2f_cml_v1.py`[3], followed the Celsius temperature:

Terminal

```
c2f_cml_v1.py 21
69.8
```

Inside the program we can fetch the text `21` as `sys.argv[1]`. The `sys` module has a list `argv` containing all the command-line arguments to the program, i.e., all the "words" appearing after the program name when we run the program. Here there is only one argument and it is stored with index `1`. The first element in the `sys.argv` list, `sys.argv[0]`, is always the name of the program.

A command-line argument is treated as a text, so `sys.argv[1]` refers to a string object, in this case `'21'`. Since we interpret the command-line argument as a number and want to compute with it, it is necessary to explicitly convert the string to a `float` object. In the program we therefore write[4]

```
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print F
```

[2] This property is quite remarkable in computer science – a string formula will in most other languages be much slower than if the formula were hardcoded inside a plain function.

[3] The `cml` part of the name is an abbreviation for "command line", and `v1` denotes "version 1", as usual.

[4] We could write `9` instead of `9.0`, in the formula for `F`, since `C` is guaranteed to be `float`, but it is safer to write `9.0`. One could think of modifying the conversion of the command-line argument to `eval(sys.argv[1])`, and in that case `C` can easily be an `int`.

As another example, consider the `ball_variables.py` program from Chapter 1.1.7. Instead of hardcoding the values of `v0` and `t` in the program we can read the two values from the command line:

Terminal

```
ball_variables2.py 0.6 5
1.2342
```

The two command-line arguments are now available as `sys.argv[1]` and `sys.argv[2]`. The complete `ball_variables2.py` program thus looks as

```
import sys
t  = float(sys.argv[1])
v0 = float(sys.argv[2])
g = 9.81
y = v0*t - 0.5*g*t**2
print y
```

Our final example here concerns a program that can add two input objects (file `add_cml.py`, corresponding to `add_input.py` from Chapter 3.1.1):

```
import sys
i1 = eval(sys.argv[1])
i2 = eval(sys.argv[2])
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

A key issue here is that we apply `eval` to the command-line arguments and thereby convert the strings into appropriate objects. Here is an example on execution:

Terminal

```
add_cml.py 2 3.1
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 5.1
```

### 3.2.2 A Variable Number of Command-Line Arguments

Let us make a program `addall.py` that adds all its command-line arguments. That is, we may run something like

Terminal

```
addall.py 1 3 5 -9.9
The sum of 1 3 5 -9.9 is -0.9
```

The command-line arguments are stored in the sublist `sys.argv[1:]`. Each element is a string so we must perform a conversion to `float` before performing the addition. There are many ways to write this program. Let us start with version 1, `addall_v1.py`:

```
import sys
s = 0
for arg in sys.argv[1:]:
    number = float(arg)
    s += number
print 'The sum of ',
for arg in sys.argv[1:]:
    print arg,
print 'is ', s
```

The output is on one line, but built of several `print` statements (note the trailing comma which prevents the usual newline, cf. page 91). The command-line arguments must be converted to numbers in the first `for` loop because we need to compute with them, but in the second loop we only need to print them and then the string representation is appropriate.

The program above can be written more compactly if desired:

```
import sys
s = sum([float(x) for x in sys.argv[1:]])
print 'The sum of %s is %s' % (' '.join(sys.argv[1:]), s)
```

Here, we convert the list `sys.argv[1:]` to a list of `float` objects and then pass this list to Python's `sum` function for adding the numbers. The construction `S.join(L)` places all the elements in the list `L` after each other with the string `S` in between. The result here is a string with all the elements in `sys.argv[1:]` and a space in between, i.e., the text that originally appeared on the command line. Chapter 6.3.1 contains more information on `join` and many other very useful string operations.

### 3.2.3 More on Command-Line Arguments

Unix commands make heavy use of command-line arguments. For example, when you write `ls -s -t` to list the files in the current folder, you run the program `ls` with two command-line arguments: `-s` and `-t`. The former specifies that `ls` shall print the file name together with the size of the file, and the latter sorts the list of files according to their dates of last modification (the most recently modified files appear first). Similarly, `cp -r my new` for copying a folder tree `my` to a new folder tree `new` invokes the `cp` program with three command line arguments: `-r` (for recursive copying of files), `my`, and `new`. Most programming languages have support for extracting the command-line arguments given to a program.

command-line arguments are separated by blanks. What if we want to provide a text containing blanks as command-line argument? The text containing blanks must then appear inside single or double quotes. Let us demonstrate this with a program that simply prints the command-line arguments:

```
import sys, pprint
pprint.pprint(sys.argv[1:])
```

Say this program is named `print_cml.py`. The execution

Terminal
```
print_cml.py 21 a string with blanks 31.3
['21', 'a', 'string', 'with', 'blanks', '31.3']
```

demonstrates that each word on the command line becomes an element in `sys.argv`. Enclosing strings in quotes, as in

Terminal
```
print_cml.py 21 "a string with blanks" 31.3
['21', 'a string with blanks', '31.3']
```

shows that the text inside the quotes becomes a single command line argument.

### 3.2.4 Option–Value Pairs on the Command Line

The examples on using command-line arguments so far require the user of the program to type all arguments in their right sequence, just as when calling a function with positional arguments. It would be very convenient to assign command-line arguments in the same way as we use keyword arguments. That is, arguments are associated with a name, their sequence can be arbitrary, and only the arguments where the default value is not appropriate need to be given. Such type of command-line arguments may have `-option value` pairs, where "option" is some name of the argument.

As usual, we shall use an example to illustrate how to work with `-option value` pairs. Consider the (hopefully well-known) physics formula for the location $s(t)$ of an object at time $t$, if the object started at $s = s_0$ at $t = 0$ with a velocity $v_0$, and thereafter was subject to a constant acceleration $a$:

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2 \,. \tag{3.1}$$

This formula requires four input variables: $s_0$, $v_0$, $a$, and $t$. We can make a program `location.py` that takes four options, `--s0`, `--v0`, `--a`, and `--t` on the command line. The program is typically run like this:

Terminal
```
location.py --t 3 --s0 1 --v0 1 --a 0.5
```

The sequence of `-option value` pairs is arbitrary.

All input variables should have sensible default values such that we can leave out the options for which the default value is suitable. For

example, if $s_0 = 0$, $v_0 = 0$, $a = 1$, and $t = 1$ by default, and we only want to change $t$, we can run

Terminal

```
location.py --t 3
```

The standard Python module `getopt` supports reading `-option value` pairs on the command line. The recipe for using `getopt` goes as follows in the present example:

```
# set default values:
s0 = v0 = 0; a = t = 1
import getopt, sys
options, args = getopt.getopt(sys.argv[1:], '',
                              ['t=', 's0=', 'v0=', 'a='])
```

Note that we specify the option names without the leading double hyphen. The trailing `=` character indicates that the option is supposed to be followed by a value (without `=` only the option and not its value can be specified on the command line – this is basically suitable for boolean variables only).

The returned `options` object is a list of (option, value) 2-tuples containing the `-option value` pairs found on the command line. For example, the `options` list may be

```
[('--v0', 1.5), ('--t', 0.1), ('--a', 3)]
```

In this case, the user specified $v_0$, $t$, and $a$, but not $s_0$ on the command line. The `args` object returned from `getopt.getopt` is a list of all the remaining command line arguments, i.e., the arguments that are not `-option value` pairs. The `args` variable has no use in our current example.

The typical way of processing the `options` list involves testing on the different option names:

```
for option, value in options:
    if option == '--t':
        t = float(value)
    elif option == '--a':
        a = float(value)
    elif option == '--v0':
        v0 = float(value)
    elif option == '--s0':
        s0 = float(value)
```

Sometimes a more descriptive options, say `--initial_velocity`, is offered in addition to the short form `-v0`. Similarly, `--initial_position` can be offered as an alternative to `-s0`. We may add as many alternative options as we like:

```
options, args = getopt.getopt(sys.argv[1:], '',
  ['v0=', 'initial_velocity=', 't=', 'time=',
   's0=', 'initial_velocity=', 'a=', 'acceleration='])
```

```
for option, value in options:
    if option in ('--t', '--time'):
        t = float(value)
    elif option in ('--a', '--acceleration'):
        a = float(value)
    elif option in ('--v0', '--initial_velocity'):
        v0 = float(value)
    elif option in ('--s0', '--initial_position'):
        s0 = float(value)
```

At this point in the program we have all input data, either by their default values or by user-given command-line arguments, and we can finalize the program by computing the formula (3.1) and printing out the result:

```
s = s0 + v0*t + 0.5*a*t**2
print """
An object, starting at s=%g at t=0 with initial
velocity %s m/s, and subject to a constant
acceleration %g m/s**2, is found at the
location s=%g m after %s seconds.
""" % (s0, v0, a, s, t)
```

A complete program using the `getopt` module as explained above is found in the file `location.py` in the `input` folder.

## 3.3 Handling Errors

Suppose we forget to provide a command-line argument to the `c2f_cml_v1.py` program from Chapter 3.2.1:

Terminal

```
c2f_cml_v1.py
Traceback (most recent call last):
  File "c2f_cml_v1.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

Python aborts the program and shows an error message containing the line where the error occured, the type of the error (`IndexError`), and a quick explanation of what the error is. From this information we deduce that the index 1 is out of range. Because there are no command-line arguments in this case, `sys.argv` has only one element, namely the program name. The only valid index is then 0.

For an experienced Python programmer this error message will normally be clear enough to indicate what is wrong. For others it would be very helpful if wrong usage could be detected by our program and a description of correct operation could be printed. The question is how to detect the error inside the program.

The problem in our sample execution is that `sys.argv` does not contain two elements (the program name, as always, plus one command-line argument). We can therefore test on the length of `sys.argv` to

detect wrong usage: if `len(sys.argv)` is less than 2, the user failed to provide information on the `C` value. The new version of the program, `c2f_cml_v1.py`, starts with this `if` test:

```
if len(sys.argv) < 2:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort because of error
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

We use the `sys.exit` function to abort the program. Any argument different from zero signifies that the program was aborted due to an error, but the precise value of the argument does not matter so here we simply choose it to be `1`. If no errors are found, but we still want to abort the program, `sys.exit(0)` is used.

A more modern and flexible way of handling potential errors in a program is to *try* to execute some statements, and if something goes wrong, the program can detect this and jump to a set of statements that handle the erroneous situation as desired. The relevant program construction reads

```
try:
    <statements>
except:
    <statements>
```

If something goes wrong when executing the statements in the `try` block, Python raises what is known as an *exception*. The execution jumps directly to the `except` block whose statements can provide a remedy for the error. The next section explains the `try-except` construction in more detail through examples.

### 3.3.1 Exception Handling

To clarify the idea of exception handling, let us use a `try-except` block to handle the potential problem arising when our Celsius-Fahrenheit conversion program lacks a command-line argument:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

The program is stored in the file `c2f_cml_v3.py`. If the command-line argument is missing, the indexing `sys.argv[1]`, which has an invalid

index 1, *raises an exception*. This means that the program jumps directly[5] to the except block. In the except block, the programmer can retrieve information about the exception and perform statements to recover from the error. In our example, we know what the error can be, and therefore we just print a message and abort the program.

Suppose the user provides a command-line argument. Now, the try block is executed successfully, and the program neglects the except block and continues with the Fahrenheit conversion. We can try out the last program in two cases:

Terminal

```
c2f_cml_v3.py
You failed to provide Celsius degrees as input on the command line!

c2f_cml_v3.py 21
21C is 69.8F
```

In the first case, the illegal index in sys.argv[1] causes an exception to be raised, and we perform the steps in the except block. In the second case, the try block executes successfully, so we jump over the except block and continue with the computations and the printout of results.

For a user of the program, it does not matter if the programmer applies an if test or exception handling to recover from a missing command-line argument. Nevertheless, exception handling is considered a better programming solution because it allows more advanced ways to abort or continue the execution. Therefore, we adopt exception handling as our standard way of dealing with errors in the rest of this book.

*Testing for a Specific Exception.* Consider the assignment

```
C = float(sys.argv[1])
```

There are two typical errors associated with this statement: i) sys.argv[1] is illegal indexing because no command-line arguments are provided, and ii) the content in the string sys.argv[1] is not a pure number that can be converted to a float object. Python detects both these errors and raises an IndexError exception in the first case and a ValueError in the second. In the program above, we jump to the except block and issue the same message regardless of what went wrong in the try block. For example, when we indeed provide a command-line argument, but write it on an illegal form (21C), the program jumps to the except block and prints a misleading message:

Terminal

```
c2f_cml_v3.py 21C
You failed to provide Celsius degrees as input on the command line!
```

[5] This implies that float is not called, and C is not initialized with a value.

The solution to this problem is to branch into different `except` blocks depending on what type of exception that was raised in the `try` block (program `c2f_cml_v4.py`):

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print 'Celsius degrees must be supplied on the command line'
    sys.exit(1)  # abort execution
except ValueError:
    print 'Celsius degrees must be a pure number, '\
          'not "%s"' % sys.argv[1]
    sys.exit(1)

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

Now, if we fail to provide a command-line argument, an `IndexError` occurs and we tell the user to write the `C` value on the command line. On the other hand, if the `float` conversion fails, because the command-line argument has wrong syntax, a `ValueError` exception is raised and we branch into the second `except` block and explain that the form of the given number is wrong:

Terminal

```
c2f_cml_v3.py 21C
Celsius degrees must be a pure number, not "21C"
```

*Examples on Exception Types.* List indices out of range lead to `IndexError` exceptions:

```
>>> data = [1.0/i for i in range(1,10)]
>>> data[9]
...
IndexError: list index out of range
```

Some programming languages (Fortran, C, C++, and Perl are examples) allow list indices outside the legal index values, and such unnoticed errors can be hard to find. Python always stops a program when an invalid index is encountered, unless you handle the exception explicitly as a programmer.

Converting a string to `float` is unsuccessful and gives a `ValueError` if the string is not a pure integer or real number:

```
>>> C = float('21 C')
...
ValueError: invalid literal for float(): 21 C
```

Trying to use a variable that is not initialized gives a `NameError` exception:

```
>>> print a
...
NameError: name 'a' is not defined
```

Division by zero rasies a `ZeroDivisionError` exception:

```
>>> 3.0/0
...
ZeroDivisionError: float division
```

Writing a Python keyword illegally or performing a Python grammar error leads to a `SyntaxError` exception:

```
>>> forr d in data:
...
    forr d in data:
         ^
SyntaxError: invalid syntax
```

What if we try to multiply a string by a number?

```
>>> 'a string'*3.14
...
TypeError: can't multiply sequence by non-int of type 'float'
```

The `TypeError` exception is raised because the object types involved in the multiplication are wrong (`str` and `float`).

*Digression.* It might come as a surprise, but multiplication of a string and a number is legal if the number is an integer. The multiplication means that the string should be repeated the specified number of times. The same rule also applies to lists:

```
>>> '--'*10   # ten double dashes = 20 dashes
'--------------------'
>>> n = 4
>>> [1, 2, 3]*n
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [0]*n
[0, 0, 0, 0]
```

The latter construction is handy when we want to create a list of `n` elements and later assign specific values to each element in a `for` loop.

### 3.3.2 Raising Exceptions

When an error occurs in your program, you may either print a message and use `sys.exit(1)` to abort the program, or you may raise an exception. The latter task is easy. You just write `raise E(message)`, where `E` can be a known exception type in Python and `message` is a string explaining what is wrong. Most often `E` means `ValueError` if the value of some variable is illegal, or `TypeError` if the type of a variable is wrong. You can also define your own exception types.

*Example.* In the program `c2f_cml_v4.py` from page 135 we show how we can test for different exceptions and abort the program. Sometimes we see that an exception may happen, but if it happens, we want a more precise error message to help the user. This can be done by raising a new exception in an `except` block and provide the desired exception type and message.

Another application of raising exceptions with tailored error messages arises when input data are invalid. The code below illustrates how to raise exceptions in various cases.

We collect the reading of `C` and handling of errors a separate function:

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError\
        ('Celsius degrees must be supplied on the command line')
    except ValueError:
        raise ValueError\
        ('Celsius degrees must be a pure number, '\
        'not "%s"' % sys.argv[1])
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```

There are two ways of using the `read_C` function. The simplest is to call the function,

```
C = read_C()
```

Wrong input will now lead to a raw dump of exceptions, e.g.,

Terminal

```
c2f_cml_v5.py
Traceback (most recent call last):
  File "c2f_cml4.py", line 5, in ?
    raise IndexError\
IndexError: Celsius degrees must be supplied on the command line
```

New users of this program may become uncertain when getting raw output from exceptions, because words like `Traceback`, `raise`, and `IndexError` do not make much sense unless you have some experience with Python. A more user-friendly output can be obtained by calling the `read_C` function inside a `try-except` block, check for any exception (or better: check for `IndexError` *or* `ValueError`), and write out the exception message in a more nicely formatted form. In this way, the programmer takes complete control of how the program behaves when errors are encountered:

```
try:
    C = read_C()
except Exception, e:
    print e             # exception message
    sys.exit(1)         # terminate execution
```

`Exception` is the parent name of all exceptions, and `e` is an exception object. Nice printout of the exception message follows from a straight `print e`. Instead of `Exception` we can write `(ValueError, IndexError)` to test more specifically for two exception types we can expect from the `read_C` function:

```
try:
    C = read_C()
except (ValueError, IndexError), e:
    print e             # exception message
    sys.exit(1)         # terminate execution
```

After the `try-except` block above, we can continue with computing `F = 9*C/5 + 32` and print out `F`. The complete program is found in the file `c2f_cml.py`. We may now test the program's behavior when the input is wrong and right:

Terminal

```
c2f_cml.py
Celsius degrees must be supplied on the command line

c2f_cml.py 21C
Celsius degrees must be a pure number, not "21C"

c2f_cml.py -500
C=-500 is a non-physical value!

c2f_cml.py 21
21C is 69.8F
```

This program deals with wrong input, writes an informative message, and terminates the execution without annoying behavior.

Scattered `if` tests with `sys.exit` calls are considered a bad programming style compared to the use of nested exception handling as illustrated above. You should abort execution in the main program only, not inside functions. The reason is that the functions can be re-used in other occasions where the error can be dealt with differently. For instance, one may avoid abortion by using some suitable default data.

The programming style illustrated above is considered the best way of dealing with errors, so we suggest that you hereafter apply exceptions for handling potential errors in the programs you make, simply because this is what experienced programmers expect from your codes.

## 3.4 A Glimpse of Graphical User Interfaces

Maybe you find it somewhat strange that the usage of the programs we have made so far in this book – and the programs we will make in the rest of the book – are less graphical and intuitive than the computer programs you are used to from school or entertainment. Those programs are operated through some self-explaning graphics, and most of the things you want to do involve pointing with the mouse, clicking on graphical elements on the screen, and maybe filling in some text fields. The programs in this book, on the other hand, are run from the command line in a terminal window or inside IPython, and input is also given here in form of plain text.

The reason why we do not equip the programs in this book with graphical interfaces for providing input, is that such graphics is both complicated and tedious to write. If the aim is to solve problems from mathematics and science, we think it is better to focus on this part rather than large amounts of code that merely offers some "expected" graphical cosmetics for putting data into the program. Textual input from the command line is also quicker to provide. Also remember that the computational functionality of a program is obviously independent from the type of user interface, textual or graphic.

As an illustration, we shall now show a Celsius to Fahrenheit conversion program with a graphical user interface (often called a GUI). The GUI is shown in Figure 3.1. We encourage you to try out the graphical interface – the name of the program is `c2f_gui.py`. The complete program text is listed below.




**Fig. 3.1** Screen dump of the graphical interface for a Celsius to Fahrenheit conversion program. The user can type in the temperature in Celsius degrees, and when clicking on the "is" button, the corresponding Fahrenheit value is displayed.

```
from Tkinter import *
root = Tk()
C_entry = Entry(root, width=4)
C_entry.pack(side='left')
Cunit_label = Label(root, text='Celsius')
Cunit_label.pack(side='left')

def compute():
    C = float(C_entry.get())
    F = (9./5)*C + 32
    F_label.configure(text='%g' % F)

compute = Button(root, text=' is ', command=compute)
compute.pack(side='left', padx=4)

F_label = Label(root, width=4)
F_label.pack(side='left')
Funit_label = Label(root, text='Fahrenheit')
```

```
Funit_label.pack(side='left')

root.mainloop()
```

The goal of the forthcoming dissection of this program is to give a taste of how graphical user interfaces are coded. The aim is not to equip you with knowledge on how you can make such programs on your own.

A GUI is built of many small graphical elements, called *widgets*. The graphical window generated by the program above and shown in Figure 3.1 has five such widgets. To the left there is an *entry* widget where the user can write in text. To the right of this entry widget is a *label* widget, which just displays some text, here "Celsius". Then we have a *button* widget, which when being clicked leads to computations in the program. The result of these computations is displayed as text in a *label* widget to the right of the button widget. Finally, to the right of this result text we have another *label* widget displaying the text "Fahrenheit". The program must construct each widget and pack it correctly into the complete window. In the present case, all widgets are packed from left to right.

The first statement in the program imports functionality from the GUI toolkit `Tkinter` to construct widgets. First, we need to make a root widget that holds the complete window with all the other widgets. This root widget is of type `Tk`. The first entry widget is then made and referred to by a variable `C_entry`. This widget is an object of type `Entry`, provided by the `Tkinter` module. Widgets constructions follow the syntax

```
variable_name = Widget_type(parent_widget, option1, option2, ...)
variable_name.pack(side='left')
```

When creating a widget, we must bind it to a *parent widget*, which is the graphical element in which this new widget is to be packed. Our widgets in the present program have the `root` widget as parent widget. Various widgets have different types of options that we can set. For example, the `Entry` widget has a possibility for setting the width of the text field, here `width=4` means that the text field is 4 characters wide. The pack statement is important to remember – without it, the widget remains invisible.

The other widgets are constructed in similar ways. The next fundamental feature of our program is how computations are tied to the event of clicking the button "is". The `Button` widget has naturally a text, but more important, it binds the button to a function `compute` through the `command=compute` option. This means that when the user clicks the button "is", the function `compute` is called. Inside the `compute` function we first fetch the Celsius value from the `C_entry` widget, using this widget's `get` function, then we transform this string (everything

typed in by the user is interpreted as text and stored in strings) to a `float` before we compute the corresponding Fahrenheit value. Finally, we can update ("configure") the text in the `Label` widget `F_label` with a new text, namely the computed degrees in Fahrenheit.

A program with a GUI behaves differently from the programs we construct in this book. First, all the statements are executed from top to bottom, as in all our other programs, but these statements just construct the GUI and define functions. No computations are performed. Then the program enters a so-called *event loop*: `root.mainloop()`. This is an infinite loop that "listens" to user events, such as moving the mouse, clicking the mouse, typing characters on the keyboard, etc. When an event is recorded, the program starts performing associated actions. In the present case, the program waits for only one event: clicking the button "is". As soon as we click on the button, the `compute` function is called and the program starts doing mathematical work. The GUI will appear on the screen until we destroy the window by click on the X up in the corner of the window decoration. More complicated GUIs will normally have a special "Quit" button to terminate the event loop.

In all GUI programs, we must first create a hierarchy of widgets to build up all elements of the user interface. Then the program enters an event loop and waits for user events. Lots of such events are registered as actions in the program when creating the widgets, so when the user clicks on buttons, move the mouse into certain areas, etc., functions in the program are called and "things happen".

Many books explain how to make GUIs in Python programs, see for instance [2, 3, 5, 7].

## 3.5 Making Modules

Sometimes you want to reuse a function from an old program in a new program. The simplest way to do this is to copy and paste the old source code into the new program. However, this is not good programming practice, because you then over time end up with multiple identical versions of the same function. When you want to improve the function or correct a bug, you need to remember to do the same update in all files with a copy of the function, and in real life most programmers fail to do so. You easily end up with a mess of different versions with different quality of basically the same code. Therefore, a golden rule of programming is to have one and only one version of a piece of code. All programs that want to use this piece of code must access one and only one place where the source code is kept. This principle is easy to implement if we create a module containing the code we want to reuse later in different programs.

You learned already in Chapter 1 how to import functions from Python modules. Now you will learn how to make your own modules. There is hardly anything to learn, because you just collect all the functions that constitute the module in one file, say with name `mymodule.py`. This file is automatically a module, with name `mymodule`, and you can import functions from this module in the standard way. Let us make everything clear in detail by looking at an example.

### 3.5.1 Example: Compund Interest Formulas

The classical formula for the growth of money in a bank reads[6]

$$A = A_0\left(1 + \frac{p}{360\cdot 100}\right)^n, \tag{3.2}$$

where $A_0$ is the initial amount of money, and $A$ is the present amount after $n$ days with $p$ percent annual interest rate. Equation (3.2) involves four parameters: $A$, $A_0$, $p$, and $n$. We may solve for any of these, given the other three:

$$A_0 = A\left(1 + \frac{p}{360\cdot 100}\right)^{-n}, \tag{3.3}$$

$$n = \frac{\ln\frac{A}{A_0}}{\ln\left(1 + \frac{p}{360\cdot 100}\right)}, \tag{3.4}$$

$$p = 360\cdot 100\left(\left(\frac{A}{A_0}\right)^{1/n} - 1\right) \tag{3.5}$$

Suppose we have implemented (3.2)–(3.5) in four functions:

```
from math import log as ln

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

We want to make these functions available in a module, say with name `interest`, so that we can import functions and compute with them in a program. For example,

[6] The formula applies the so-called Actual/360 convention where the rate per day is computed as $p/360$, while $n$ counts the actual number of days the money is in the bank. See "Day count convention" in Wikipedia for detailed information and page 238 for a Python module for computing the number of days between two dates.

```
from interest import days
A0 = 1; A = 2; p = 5
n = days(A0, 2, p)
years = n/365.0
print 'Money has doubled after %.1f years' % years
```

How to make the `interest` module is described next.

### 3.5.2 Collecting Functions in a Module File

To make a module of the four functions `present_amount`, `initial_amount`, `days`, and `annual_rate`, we simply open an empty file in a text editor and copy the program code for all the four functions over to this file. This file is then automatically a Python module provided we save the file under any valid filename. The extension must be `.py`, but the module name is only the base part of the filename. In our case, the filename `interest.py` implies a module name `interest`. To use the `annual_rate` function in another program we simply write, in that program file,

```
from interest import annual_rate
```

or we can write

```
from interest import *
```

to import all four functions, or we can write

```
import interest
```

and access individual functions as `interest.annual_rate` and so forth.

*Test Block.* It is recommended to only have functions and not any statements outside functions in a module[7]. However, Python allows a special construction to let the file act both as a module with function definitions only *and* as an ordinary program that we can run, i.e., with statements that apply the functions and possibly write output. This two-fold "magic" consists of putting the application part after an `if` test of the form

```
if __name__ == '__main__':
    <block of statements>
```

The `__name__` variable is automatically defined in any module and equals the module name if the module file is imported in another program, or `__name__` equals the string `'__main__'` if the module file is

[7] The module file is executed from top to bottom during the import. With function definitions only in the module file, there will be no calculations or output from the import, just definitions of functions. This is the desirable behavior.

run as a program. This implies that the `<block of statements>` part is executed if and only if we run the module file as a program. We shall refer to `<block of statements>` as the *test block* of a module.

Often, when modules are created from an ordinary program, the original main program is used as test block. The new module file then works as the old program, but with the new possibility of being imported in other programs. Let us write a little main program for testing the `interest` module. The idea is that we assign compatible values to the four parameters and check that given three of them, the functions calculate the remaining parameter in the correct way:

```
if __name__ == '__main__':
    A = 2.2133983053266699
    A0 = 2.0
    p = 5
    n = 730
    print 'A=%g (%g)\nA0=%g (%.1f)\nn=%d (%d)\np=%g (%.1f)' % \
          (present_amount(A0, p, n), A,
           initial_amount(A, p, n), A0,
           days(A0, A, p), n,
           annual_rate(A0, A, n), p)
```

Running the module file as a program is now possible:

Terminal

```
interest.py
A=2.2134 (2.2134)
A0=2 (2.0)
n=730 (730)
p=5 (5.0)
```

The computed values appear after the equal sign, with correct values in parenthesis. We see that the program works well.

To test that the `interest.py` also works as a module, invoke a Python shell and try to import a function and compute with it:

```
>>> from interest import present_amount
>>> present_amount(2, 5, 730)
2.2133983053266699
```

We have therefore demonstrated that the file `interest.py` works both as a program and as a module.

*Flexible Test Blocks.* It is a good programming practice to let the test block do one or more of three things: (i) provide information on how the module or program is used, (ii) test if the module functions work properly, and (iii) offer interaction with users such that the module file can be applied as a useful program.

Instead of having a lot of statements in the test block, it might be better to collect the statements in separate functions, which then are called from the test block. A convention is to let these test or documentation functions have names starting with an underscore, because such

names are not imported in other programs when doing a `from module import *` (normally we do not want to import test or documentation functions). In our example we may collect the verification statements above in a separate function and name this function `_verify` (observe the leading underscore). We also write the code a bit more explicit to better demonstrate how the module functions can be used:

```
def _verify():
    # compatible values:
    A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
    # given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis):
    A_computed = present_amount(A0, p, n)
    A0_computed = initial_amount(A, p, n)
    n_computed = days(A0, A, p)
    p_computed = annual_rate(A0, A, n)
    print 'A=%g (%g)\nA0=%g (%.1f)\nn=%d (%d)\np=%g (%.1f)' % \
          (A_computed, A, A0_computed, A0,
           n_computed, n, p_computed, p)
```

We may require a single command-line argument `verify` to run the verification. The test block can then be expressed as

```
if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == 'verify':
        _verify()
```

To make a useful program, we may allow setting three parameters on the command line and let the program compute the remaining parameter. For example, running the program as

Terminal

```
interest.py A0=2 A=1 n=1095
```

should lead to a computation of $p$, in this case for seeing the size of the annual interest rate if the amount is to be doubled after three years.

How can we achieve the desired functionality? Since variables are already introduced and “initialized” on the command line, we could grab this text and execute it as Python code, either as three different lines or with semicolon between each assignment. This is easy[8]:

```
init_code = ''
for statement in sys.argv[1:]:
    init_code += statement + '\n'
exec(init_code)
```

For the sample run above with `A0=2 A=1 n=1095` on the command line, `init_code` becomes the string

```
A0=2
A=1
n=1095
```

[8] The `join` function on page 295 in Chapter 6.3.1, see also page 129, is more elegant and avoids the loop.

Note that one cannot have spaces around the equal signs on the command line as this will break an assignment like `A0 = 2` into three command-line arguments, which will give rise to a `SyntaxError` in `exec(init_code)`. To tell the user about such errors, we execute `init_code` inside a `try-except` block:

```
try:
    exec(init_code)
except SyntaxError, e:
    print e
    print init_code
    sys.exit(1)
```

At this stage, our program has hopefully initialized three parameters in a successful way, and it remains to detect the remaining parameter to be computed. The following code does the work:

```
if 'A=' not in init_code:
    print 'A =', present_amount(A0, p, n)
elif 'A0=' not in init_code:
    print 'A0 =', initial_amount(A, p, n)
elif 'n=' not in init_code:
    print 'n =', days(A0, A , p)
elif 'p=' not in init_code:
    print 'p =', annual_rate(A0, A, n)
```

It may happen that the user of the program assign value to a parameter with wrong name or forget a parameter. In those cases we call one of our four functions with uninitialized arguments. Therefore, we should embed the code above in a `try-except` block. An unitialized variable will lead to a `NameError`, while another frequent error is illegal values in the computations, leading to a `ValueError` exception. It is also a good habit to collect all the code related to computing the remaining, fourth parameter in a function for separating this piece of code from other parts of the module file:

```
def _compute_missing_parameter(init_code):
    try:
        exec(init_code)
    except SyntaxError, e:
        print e
        print init_code
        sys.exit(1)
    # find missing parameter:
    try:
        if 'A=' not in init_code:
            print 'A =', present_amount(A0, p, n)
        elif 'A0=' not in init_code:
            print 'A0 =', initial_amount(A, p, n)
        elif 'n=' not in init_code:
            print 'n =', days(A0, A , p)
        elif 'p=' not in init_code:
            print 'p =', annual_rate(A0, A, n)
    except NameError, e:
        print e
        sys.exit(1)
    except ValueError:
```

```
        print 'Illegal values in input:', init_code
        sys.exit(1)
```

If the user of the program fails to give any command-line arguments, we print a usage statement. Otherwise, we run a verification if the first command-line argument is "verify", and else we run the missing parameter computation (i.e., the useful main program):

```
_filename = sys.argv[0]
_usage = """
Usage: %s A=10 p=5 n=730
Program computes and prints the 4th parameter'
(A, A0, p, or n)""" % _filename

if __name__ == '__main__':
    if len(sys.argv) == 1:
        print _usage
    elif len(sys.argv) == 2 and sys.argv[1] == 'verify':
        _verify()
    else:
        init_code = ''
        for statement in sys.argv[1:]:
            init_code += statement + '\n'
        _compute_missing_parameter(init_code)
```

Note leading underscores in variable names that are to be used locally in the `interest.py` file only.

It is also a good habit to include a doc string in the beginning of the module file. This doc string explains the purpose and use of the module:

```
"""
Module for computing with interest rates.
Symbols: A is present amount, A0 is initial amount,
n counts days, and p is the interest rate per year.

Given three of these parameters, the fourth can be
computed as follows:

    A  = present_amount(A0, p, n)
    A0 = initial_amount(A, p, n)
    n  = days(A0, A, p)
    p  = annual_rate(A0, A, n)
"""
```

You can run the `pydoc` program to see a documentation of the new module, containing the doc string above and a list of the functions in the module: just write `pydoc interest` in a terminal window.

Now the reader is recommended to take a look at the actual file `interest.py` in `src/input` to see all elements of a good module file at once: doc string, set of functions, verification function, "main program function", usage string, and test block.

### 3.5.3 Using Modules

Let us further demonstrate how to use the `interest.py` module in programs. For illustration purposes, we make a separate program file, say with name `test.py`, containing some computations:

```
from interest import days

# how many days does it take to double an amount when the
# interest rate is p=1,2,3,...14?
for p in range(1, 15):
    years = days(1, 2, p)/365.0
    print 'With p=%d%% it takes %.1f years to double the amount' \
    % (p, years)
```

There are different ways to import functions in a module, and let us explore these in an interactive session. The function call `dir()` will list all names we have defined, including imported names of variables and functions. Calling `dir(m)` will print the names defined inside a module with name `m`. First we start an interactive shell and call `dir()`

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

These variables are always defined. Running the IPython shell will introduce several other standard variables too. Doing

```
>>> from interest import *
>>> dir()
[ ..., 'annual_rate', 'days', 'initial_amount',
'present_amount', 'ln', 'sys']
```

shows that we get our four functions imported, along with `ln` and `sys`. The latter two are needed in the `interest` module, but not necessarily in our new program `test.py`. Observe that none of the names with a leading underscore are imported. This demonstrates the importance of using a leading underscore in names for local variables and functions in a module: Names local to a module will then not pollute other programs or interactive sessions when a "star import" (`from module import *`) is performed.

Next we do

```
>>> import interest
>>> dir(interest)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '_compute_missing_parameter', '_usage', '_verify',
 'annual_rate', 'days', 'filename', 'initial_amount',
 'ln', 'present_amount', 'sys']
```

All variables and functions defined or imported in the `interest.py` file are now visible, and we can access also functions and variables beginning with an underscore as long as we have the `interest.` prefix:

```
>>> interest._verify()
A=2.2134 (2.2134)
A0=2 (2.0)
n=730 (730)
p=5 (5.0)
>>> interest._filename
```

The `test.py` program works well as long as it is located in the same folder as the `interest.py` module. However, if we move `test.py` to another folder and run it, we get an error:

Terminal

```
test.py
Traceback (most recent call last):
  File "tmp.py", line 1, in <module>
    from interest import days
ImportError: No module named interest
```

Unless the module file resides in the same folder, we need to tell Python where to find our module. Python looks for modules in the folders contained in the list `sys.path`. A little program

```
import sys, pprint
pprint.pprint(sys.path)
```

prints out all these predefined module folders. You can now do one of two things:

1. Place the module file in one of the folders in `sys.path`.
2. Include the folder containing the module file in `sys.path`.

There are two ways of doing the latter task:

2a. You can explicitly insert a new folder name in `sys.path` in the program that uses the module[9]:

```
modulefolder = '../../pymodules'
sys.path.insert(0, modulefolder)
```

Python searches the folders in the sequence they appear in the `sys.path` list so by inserting the folder name as the first list element we ensure that our module is found quickly, and in case there are other modules with the same name in other folders in `sys.path`, the one in `modulefolder` gets imported.

2b. Your module folders can be permanently specified in the `PYTHONPATH` environment variable[10]. All folder names listed in `PYTHONPATH` are automatically included in `sys.path` when a Python program starts.

[9] In this sample path, the slashes are Unix specific. On Windows you must use backward slashes and a raw string. A better solution is to express the path as `os.path.join(os.pardir, os.pardir, 'mymodules')`. This will work on all platforms.

[10] This makes sense only if you know what environment variables are, and we do not intend to explain that at the present stage.

## 3.6 Summary

### 3.6.1 Chapter Topics

*Question and Answer Input.* Prompting the user and reading the answer back into a variable is done by

```
var = raw_input('Give value: ')
```

The `raw_input` function returns a string containing the characters that the user wrote on the keyboard before pressing the Return key. It is necessary to convert `var` to an appropriate object (`int` or `float`, for instance) if we want to perform mathematical operations with `var`. Sometimes

```
var = eval(raw_input('Give value: '))
```

is a flexible and easy way of transforming the string to the right type of object (integer, real number, list, tuple, and so on). This last statement will not work, however, for strings unless the text is surrounded by quotes when written on the keyboard. A general conversion function that turns any text without quotes into the right object is `scitools.misc.str2obj`:

```
from scitools.misc import str2obj
var = str2obj(raw_input('Give value: '))
```

Typing, for example, `3` makes `var` refer to an `int` object, `3.14` results in a `float` object, `[-1,1]` results in a `list`, `(1,3,5,7)` in a `tuple`, and `some text` in the string (`str`) object `'some text'` (run the program `str2obj_demo.py` to see this functionality demonstrated).

*Getting Command-Line Arguments.* The `sys.argv[1:]` list contains all the command-line arguments given to a program (`sys.argv[0]` contains the program name). All elements in `sys.argv` are strings. A typical usage is

```
parameter1 = float(sys.argv[1])
parameter2 = int(sys.argv[2])
parameter3 = sys.argv[3]          # parameter3 can be string
```

*Using Option-Value Pairs.* Python has two modules, `getopt` and `optparse`, for interpreting command-line arguments of the form `-option value`. A simple recipe with `getopt` reads

```
import getopt, sys
try:
    options, args = getopt.getopt(sys.argv[1:], '',
           ['parameter1=', 'parameter2=', 'parameter3=',
            'p1=', 'p2=', 'p3='] # shorter forms
```

```
except getopt.GetoptError, e:
    print 'Error in command-line option:\n', e
    sys.exit(1)

# set default values:
parameter1 = ...
parameter2 = ...
parameter3 = ...

from scitools.misc import str2obj
for option, value in options:
    if option in ('--parameter1', '--p1'):
        parameter1 = eval(value)     # if not string
    elif option in ('--parameter2', '--p2'):
        parameter2 = value           # if string
    elif option in ('--parameter3', '--p3'):
        parameter3 = str2obj(value)  # any object
```

On the command line we can provide any or all of these options:

```
--parameter1 --p1 --parameter2 --p2 --parameter3 --p3
```

Each option must be succeeded by a suitable value.

*Generating Code on the Fly.* Calling `eval(s)` turns a string `s`, containing a Python expression, into code as if the contents of the string were written directly into the program code. The result of the following `eval` call is a `float` object holding the number 21.1:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

The `exec` function takes a string with arbitrary Python code as argument and executes the code. For example, writing

```
exec("""
def f(x):
    return %s
""" % sys.argv[1])
```

is the same as if we had hardcoded the (for the programmer unknown) contents of `sys.argv[1]` into a function definition in the program.

*Turning String Formulas into Python Functions.* Given a mathematical formula as a string, `s`, we can turn this formula into a callable Python function `f(x)` by

```
from scitools.StringFunction import StringFunction
# or
from scitools.std import *

f = StringFunction(s)
```

The string formula can contain parameters and an independent variable with another name than `x`:

```
Q_formula = 'amplitude*sin(w*t-phaseshift)'
Q = StringFunction(Q_formula, independent_variable='t',
                   amplitude=1.5, w=pi, phaseshift=0)
values1 = [Q(i*0.1) for t in range(10)]
Q.set_parameters(phaseshift=pi/4, amplitude=1)
values2 = [Q(i*0.1) for t in range(10)]
```

Functions of several independent variables are also supported:

```
f = StringFunction('x+y**2+A', independent_variables=('x', 'y'),
                   A=0.2)
x = 1; y = 0.5
print f(x, y)
```

*Handling Exceptions.* Testing for potential errors is done with `try-except` blocks:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

The most common exception types are `NameError` for an undefined variable, `TypeError` for an illegal value in an operation, and `IndexError` for a list index out of bounds.

*Raising Exceptions.* When some error is encountered in a program, the programmer can raise an exception:

```
if z < 0:
    raise ValueError('z=%s is negative - cannot do log(z)' % z)
r = log(z)
```

*Modules.* A module is created by putting a set of functions in a file. The filename (minus the required extension `.py`) is the name of the module. Other programs can import the module only if it resides in the same folder or in a folder contained in the `sys.path` list (see Chapter 3.5.3 for how to deal with this potential problem). Optionally, the module file can have a special `if` construct at the end, called test block, which tests the module or demonstrates its usage. The test block does not get executed when the module is imported in another program, only when the module file is run as a program.

### 3.6.2 Summarizing Example: Bisection Root Finding

*Problem.* The summarizing example of this chapter concerns the implementation of the Bisection method for solving nonlinear equations of the form $f(x) = 0$ with respect to $x$. For example, the equation

$$x = 1 + \sin x$$

can be cast to the form $f(x) = 0$ if we move all terms to the left-hand side and define $f(x) = x - 1 - \sin x$. We say that $x$ is a *root* of the equation $f(x) = 0$ if $x$ is a solution of this equation. Nonlinear equations $f(x) = 0$ can have zero, one, many, or infinitely many roots.

Numerical methods for computing roots normally lead to approximate results only, i.e., $f(x)$ is not made exactly zero, but very close to zero. More precisely, an approximate root $x$ fulfills $|f(x)| \leq \epsilon$, where $\epsilon$ is a small number. Methods for finding roots are of an iterative nature: We start with a rough approximation to a root and perform a repetive set of steps that aim to improve the approximation. Our particular method for computing roots, the Bisection method, guarantees to find an approximate root, while other methods, such as the widely used Newton's method (see Chapter 5.1.9), can fail to find roots.

The idea of the Bisection method is to start with an interval $[a, b]$ that contains a root of $f(x)$. The interval is halved at $m = (a + b)/2$, and if $f(x)$ changes sign in the left half interval $[a, m]$, one continues with that interval, otherwise one continues with the right half interval $[m, b]$. This procedure is repeated, say $n$ times, and the root is then guaranteed to be inside an interval of length $2^{-n}(b - a)$. The task is to write a program that implements the Bisection method and verify the implementation.

*Solution.* To implement the Bisection method, we need to translate the description in the previous paragraph to a precise algorithm that can be almost directly translated to computer code. Since the halving of the interval is repeated many times, it is natural to do this inside a loop. We start with the interval $[a, b]$, and adjust $a$ to $m$ if the root must be in the right half of the interval, or we adjust $b$ to $m$ if the root must be in the left half. In a language close to computer code we can express the algorithm precisely as follows:

for $i = 0, 1, 2, \ldots, n$
    $m = (a + b)/2$
    if $f(a)f(m) \leq 0$ then
        $b = m$ (root is in left half)
    else
        $a = m$ (root is in right half)
    end if
end for
$f(x)$ has a root in $[a, b]$

Figure 3.2 displays graphically the first four steps of this algorithm for solving the equation $cos(\pi x) = 0$, starting with the interval $[0, 0.82]$. The graphs are automatically produced by the program

bisection_movie.py, which was run as follows for this particular example:

Terminal
```
bisection_movie.py 'cos(pi*x)' 0 0.82
```

The first command-line argument is the formula for $f(x)$, the next is $a$, and the final is $b$.

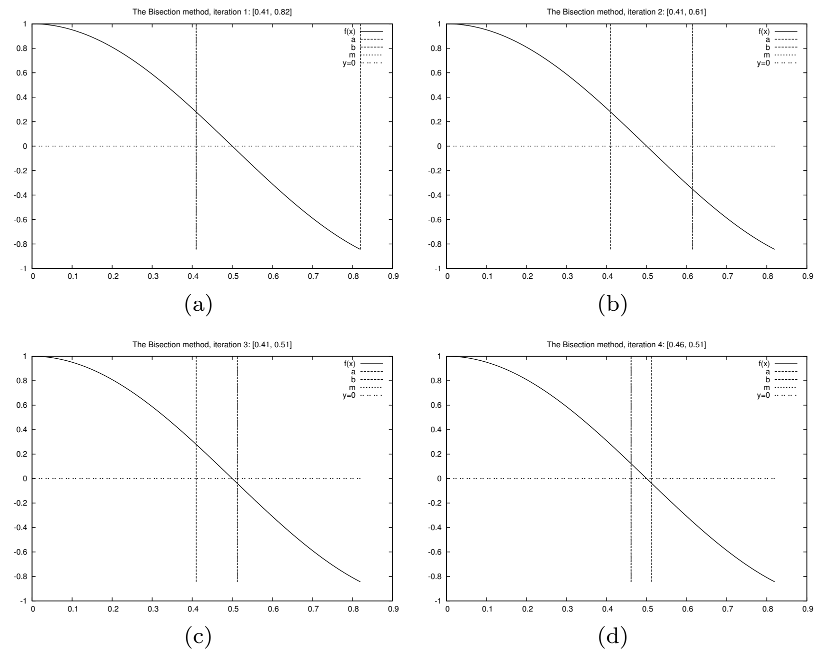



**Fig. 3.2** Illustration of the first four iterations of the Bisection algorithm for solving $\cos(\pi x) = 0$. The vertical lines correspond to the current value of $a$ and $b$.

In the algorithm listed above, we recompute $f(a)$ in each `if`-test, but this is not necessary if $a$ has not changed since the last $f(a)$ computations. It is a good habit in numerical programming to avoid redundant work. On modern computers the Bisection algorithm normally runs so fast that we can afford to do more work than necessary. However, if $f(x)$ is not a simple formula, but computed by comprehensive calculations in a program, the evaluation of $f$ might take minutes or even hours, and reducing the number of evaluations in the Bisection algorithm is then very important. We will therefore introduce extra variables in the algorithm above to save an $f(m)$ evaluation in each iteration in the `for` loop:

$f_a = f(a)$
for $i = 0, 1, 2, \ldots, n$
  $m = (a + b)/2$
  $f_m = f(m)$
  if $f_a f_m \leq 0$ then
    $b = m$ (root is in left half)
  else
    $a = m$ (root is in right half)
    $f_a = f_m$
  end if
end for
$f(x)$ has a root in $[a, b]$

To execute the algorithm above, we need to specify $n$. Say we want to be sure that the root lies in an interval of maximum extent $\epsilon$. After $n$ iterations the length of our current interval is $2^{-n}(b - a)$, if $[a, b]$ is the initial interval. The current interval is sufficiently small if

$$2^{-n}(b - a) = \epsilon,$$

which implies

$$n = -\frac{\ln \epsilon - \ln(b - a)}{\ln 2} \,. \tag{3.6}$$

Instead of calculating this $n$, we may simply stop the iterations when the length of the current interval is less than $\epsilon$. The loop is then naturally implemented as a `while` loop testing on whether $b - a \leq \epsilon$. To make the algorithm more foolproof, we also insert a test to ensure that $f(x)$ really changes sign in the initial interval[11].

Our final version of the Bisection algorithm now becomes

[11] This guarantees a root in $[a, b]$. However, $f(a)f(b) < 0$ is not a necessary condition if there is an even number of roots in the initial interval.

$f_a = f(a)$
if $f_a f(b) > 0$ then
  error: $f$ does not change sign in $[a, b]$
end if
$i = 0$ (iteration counter)
while $b - a > \epsilon$:
  $i \leftarrow i + 1$
  $m = (a + b)/2$
  $f_m = f(m)$
  if $f_a f_m \leq 0$ then
    $b = m$ (root is in left half)
  else
    $a = m$ (root is in right half)
    $f_a = f_m$
  end if
end while
if $x$ is the real root, $|x - m| < \epsilon$

This is the algorithm we aim to implement in a Python program.

A direct translation of the previous algorithm to a Python program should be quite a simple process:

```
eps = 1E-5
a, b = 0, 10

fa = f(a)
if fa*f(b) > 0:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
    sys.exit(1)

i = 0   # iteration counter
while b-a > eps:
    i += 1
    m = (a + b)/2.0
    fm = f(m)
    if fa*fm <= 0:
        b = m  # root is in left half of [a,b]
    else:
        a = m  # root is in right half of [a,b]
        fa = fm
    print 'Iteration %d: interval=[%g, %g]' % (i, a, b)

x = m          # this is the approximate root
print 'The root is', x, 'found in', i, 'iterations'
print 'f(%g)=%g' % (x, f(x))
```

This program is found in the file `bisection_v1.py`.

*Verification.* To verify the implementation in `bisection_v1.py` we choose a very simple $f(x)$ where we know the exact root. One suitable example is a linear function, $f(x) = 2x - 3$ such that $x = 3/2$ is the root of $f$. As can be seen from the source code above, we have inserted a `print` statement inside the `while` loop to control that the

program really does the right things. Running the program yields the output

```
Iteration 1: interval=[0, 5]
Iteration 2: interval=[0, 2.5]
Iteration 3: interval=[1.25, 2.5]
Iteration 4: interval=[1.25, 1.875]
...
Iteration 19: interval=[1.5, 1.50002]
Iteration 20: interval=[1.5, 1.50001]
The root is 1.50000572205 found in 20 iterations
f(1.50001)=1.14441e-05
```

It seems that the implementation works. Further checks should include hand calculations for the first (say) three iterations and comparison of the results with the program.

*Making a Function.* The previous implementation of the bisection algorithm is fine for many purposes. To solve a new problem $f(x) = 0$ it is just necessary to change the `f(x)` function in the program. However, if we encounter solving $f(x) = 0$ in another program in another context, we must put the bisection algorithm into that program in the right place. This is simple in practice, but it requires some careful work, and it is easy to make errors. The task of solving $f(x) = 0$ by the bisection algorithm is much simpler and safer if we have that algorithm available as a function in a module. Then we can just import the function and call it. This requires a minimum of writing in later programs.

When you have a "flat" program as shown above, without basic steps in the program collected in functions, you should always consider dividing the code into functions. The reason is that parts of the program will be much easier to reuse in other programs. You save coding, and that is a good rule! A program with functions is also easier to understand, because statements are collected into logical, separate units, which is another good rule! In a mathematical context, functions are particularly important since they naturally split the code into general algorithms (like the bisection algorithm) and a problem-specific part (like a special choice of $f(x)$).

Shuffling statements in a program around to form a new and better designed version of the program is called *refactoring*. We shall now refactor the `bisection_v1.py` program by putting the statements in the bisection algorithm in a function `bisection`. This function naturally takes $f(x)$, $a$, $b$, and $\epsilon$ as parameters and returns the found root, perhaps together with the number of iterations required:

```
def bisection(f, a, b, eps):
    fa = f(a)
    if fa*f(b) > 0:
        return None, 0

    i = 0   # iteration counter
    while b-a < eps:
        i += 1
        m = (a + b)/2.0
        fm = f(m)
```

```
        if fa*fm <= 0:
            b = m  # root is in left half of [a,b]
        else:
            a = m  # root is in right half of [a,b]
            fa = fm
    return m, i
```

After this function we can have a test program:

```
def f(x):
    return 2*x - 3   # one root x=1.5

eps = 1E-5
a, b = 0, 10
x, iter = bisection(f, a, b, eps)
if x is None:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
else:
    print 'The root is', x, 'found in', iter, 'iterations'
    print 'f(%g)=%g' % (x, f(x))
```

The complete code is found in file `bisection_v2.py`.

*Making a Module.* A motivating factor for implementing the bisection algorithm as a function `bisection` was that we could import this function in other programs to solve $f(x) = 0$ equations. However, if we do an import

```
from bisection_v2 import bisection
```

the import statement will run the main program in `bisection_v2.py`. We do not want to solve a particular $f(x) = 0$ example when we do an import of the `bisection` function! Therefore, we must put the main program in a test block (see Chapter 3.5.2). Even better is to collect the statements in the test program in a function and just call this function from the test block:

```
def _test():
    def f(x):
        return 2*x - 3   # one root x=1.5

    eps = 1E-5
    a, b = 0, 10
    x, iter = bisection(f, a, b, eps)
    if x is None:
        print 'f(x) does not change sign in [%g,%g].' % (a, b)
    else:
        print 'The root is', x, 'found in', iter, 'iterations'
        print 'f(%g)=%g' % (x, f(x))

if __name__ == '__main__':
    _test()
```

The complete module with the `bisection` function, the `_test` function, and the test block is found in the file `bisection.py`.

*Using the Module.* Suppose you want to solve $x = \sin x$ using the `bisection` module. What do you have to do? First, you reformulate

the equation as $f(x) = 0$, i.e., $x - \sin x = 0$ so that you identify $f(x) = x - \sin x$. Second, you make a file, say `x_eq_sinx.py`, where you import the `bisection` function, define the $f(x)$ function, and call `bisection`:

```
from bisection import bisection
from math import sin

def f(x):
    return x - sin(x)

root, iter = bisection(f, -2, 2, 1E-6)
print root
```

*A Flexible Program for Solving $f(x) = 0$.* The previous program hard-codes the input data $f(x)$, $a$, $b$, and $\epsilon$ to the bisection method for a specific equation. As we have pointed out in this chapter, a better solution is to let the user provide input data while the program is running. This approach avoids editing the program when a new equation needs to be solved (and as you remember, any change in a program has the danger of introducing new errors). We therefore set out to create a program that reads $f(x)$, $a$, $b$, and $\epsilon$ from the command-line. The expression for $f(x)$ is given as a text and turned into a Python function with aid of the `StringFunction` object from Chapter 3.1.4. The other parameters – $a$, $b$, and $\epsilon$ – can be read directly from the command line, but it can be handy to allow the user not to specify $\epsilon$ and provide a default value in the program instead.

The ideas above can be realized as follows in a new, general program for solving $f(x) = 0$ equations. The program is called `bisection_solver.py`:

```
import sys
usage = '%s f-formula a b [epsilon]' % sys.argv[0]
try:
    f_formula = sys.argv[1]
    a = float(sys.argv[2])
    b = float(sys.argv[3])
except IndexError:
    print usage; sys.exit(1)

try:  # is epsilon given on the command-line?
    epsilon = float(sys.argv[4])
except IndexError:
    epsilon = 1E-6  # default value

from scitools.StringFunction import StringFunction
from math import *  # might be needed for f_formula
f = StringFunction(f_formula)
from bisection import bisection

root, iter = bisection(f, a, b, epsilon)
if root == None:
    print 'The interval [%g, %g] does not contain a root' % (a, b)
    sys.exit(1)
print 'Found root %g\nof %s = 0 in [%g, %g] in %d iterations' % \
      (root, f_formula, a, b, iter)
```

Let us solve

1. $x = \tanh x$ with start interval $[-10, 10]$ and default precision ($\epsilon = 10^{-6}$),
2. $x^5 = \tanh(x^5)$ with start interval $[-10, 10]$ and default precision.

Both equations have one root $x = 0$.

Terminal

```
bisection_solver.py "x-tanh(x)" -10 10
Found root -5.96046e-07
of x-tanh(x) = 0 in [-10, 10] in 25 iterations

bisection_solver.py "x**5-tanh(x**5)" -10 10
Found root -0.0266892
of x**5-tanh(x**5) = 0 in [-10, 10] in 25 iterations
```

These results look strange. In both cases we halve the start interval $[-10, 10]$ 25 times, but in the second case we end up with a much less accurate root although the value of $\epsilon$ is the same. A closer inspection of what goes on in the bisection algorithm reveals that the inaccuracy is caused by round-off errors. As $a, b, m \to 0$, raising a small number to the fifth power in the expression for $f(x)$ yields a much smaller result. Subtracting a very small number $\tanh x^5$ from another very small number $x^5$ may result in a small number with wrong sign, and the sign of $f$ is essential in the bisection algorithm. We encourage the reader to graphically inspect this behavior by running these two examples with the `bisection_plot.py` program using a smaller interval $[-1, 1]$ to better see what is going on. The command-line arguments for the `bisection_plot.py` program are `'x-tanh(x)' -1 1` and `'x**5-tanh(x**5)' -1 1`. The very flat area, in the latter case, where $f(x) \approx 0$ for $x \in [-1/2, 1/2]$ illustrates well that it is difficult to locate an exact root.

## 3.7 Exercises

**Exercise 3.1.** *Make an interactive program.*

Make a program that (i) asks the user for a temperature in Fahrenheit and reads the number; (ii) computes the correspodning temperature in Celsius degrees; and (iii) prints out the temperature in the Celsius scale. Name of program file: `f2c_qa.py`. ◇

**Exercise 3.2.** *Read from the command line in Exer. 3.1.*

Modify the program from Exercise 3.1 such that the Fahrenheit temperature is read from the command line. Name of program file: `f2c_cml.py`. ◇

**Exercise 3.3.** *Use exceptions in Exer. 3.2.*

Extend the program from Exercise 3.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Name of program file: `f2c_cml.py`. ◇

**Exercise 3.4.** *Read input from the keyboard.*

Make a program that asks the user for an integer, a real number, a list, a tuple, and a string. Use `eval` to convert the input string to a list or tuple. Name of program file: `objects_qa1.py`. ◇

**Exercise 3.5.** *Read input from the command line.*

Let a program store the result of applying the `eval` function to the first command-line argument. Print out the resulting object and its type (use `type` from Chapter 1.5.2). Run the program with different input: an integer, a real number, a list, and a tuple. Then try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy? Name of program file: `objects_cml.py`. ◇

**Exercise 3.6.** *Prompt the user for input to the formula (1.1).*

Consider the simplest program for evaluting (1.1):

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Modify this code so that the program asks the user questions `t=?` and `v0=?`, and then gets `t` and `v0` from the user's input through the keyboard. Name of program file: `ball_qa.py`. ◇

**Exercise 3.7.** *Read command line input for the formula (1.1).*

Modify the program listed in Exercise 3.6 such that `v0` and `t` are read from the command line. Name of program file: `ball_cml.py`. ◇

**Exercise 3.8.** *Make the program from Exer. 3.7 safer.*

The program from Exercise 3.7 reads input from the command line. Extend that program with exception handling such that missing command-line arguments are detected. In the `except IndexError` block, use the `raw_input` function to ask the user for missing input data. Name of program file: `ball_cml_qa.py`. ◇

**Exercise 3.9.** *Test more in the program from Exer. 3.7.*

Test if the `t` value read in the program from Exercise 3.7 lies between 0 and $\frac{2v_0}{g}$. If not, print a message and abort execution. Name of program file: `ball_cml_errorcheck.py`. ◇

**Exercise 3.10.** *Raise an exception in Exer. 3.9.*

Instead of printing an error message and aborting the program explicitly, raise a `ValueError` exception in the `if` test on legal `t` values in

the program from Exercise 3.9. The exception message should contain the legal interval for $t$. Name of program file: `ball_cml_ValueError.py`. ◇

**Exercise 3.11.** *Look up calendar functionality.*

The purpose of this exercise is to make a program which takes a date, consisting of year (4 digits), month (2 digits), and day (1-31) on the command line and prints the corresponding name of the weekday (Monday, Tuesday, etc.). Python has a module `calendar`, which you must look up in the Python Library Reference (see Chapter 2.4.3), for calculating the weekday of a date. Name of program file: `weekday.py`. ◇

**Exercise 3.12.** *Use the StringFunction tool.*

Make the program `user_formula.py` from Chapter 3.1.3 shorter by using the convenient `StringFunction` tool from Chapter 3.1.4. Name of program file: `user_formula2.py`. ◇

**Exercise 3.13.** *Extend a program from Ch. 3.2.1.*

How can you modify the `add_cml.py` program from the end of Chapter 3.1.2 such that it accepts input like `sqrt(2)` and `sin(1.2)`? In this case the output should be

```
<type 'float'> + <type 'float'> becomes <type 'float'>
with value 2.34625264834
```

(Hint: Mathematical functions, such as `sqrt` and `sin`, must be defined in the program before using `eval`. Furthermore, Unix (`bash`) does not like the parentheses on the command line so you need to put quotes around the command-line arguments.) Name of program file: `add2.py`. ◇

**Exercise 3.14.** *Why we test for specific exception types.*

The simplest way of writing a `try-except` block is to test for any exception, for example,

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command-line argument'
    sys.exit(1)
```

Write the above statements in a program and test the program. What is the problem?

The fact that a user can forget to supply a command-line argument when running the program was the original reason for using a `try` block. Find out what kind of exception that is relevant for this error and test for this specific exception and re-run the program. What is the problem now? Correct the program. Name of program file: `cml_exception.py`. ◇

**Exercise 3.15.** *Make a simple module.*

Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: `C2F`, `F2C`, `C2K`, `K2C`, `F2K`, and `K2F`. Collect these functions in a module `convert_temp`. Make some sample calls to these functions from an interactive Python shell. Name of program file: `convert_temp.py`. ◇

**Exercise 3.16.** *Make a useful main program for Exer. 3.15.*

Extend the module made in Exercise 3.15 with a main program in the test block. This main program should read the first command-line argument as a numerical value of a temperature and the second argument as a temperature scale: `C`, `K`, or `F`. Write out the temperature in the other two scales. For example, if `21.3 C` is given on the command line, the output should be `70.34 F 294.45 K`. Name of program file: `convert_temp2.py`. ◇

**Exercise 3.17.** *Make a module in Exer. 2.39.*

Collect the functions in the program from Exercise 2.39 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program) in a separate function `table(t, T, n_values)`, and call this function only if the module file is run as a program (i.e., include a test block, see Chapter 3.5.2). Name of program file: `compute_sum_S_module.py`. ◇

**Exercise 3.18.** *Extend the module from Exer. 3.17.*

Extend the program from Exercise 3.17 such that $t$, $T$, and $n$ are read from the command line. The extended program should import the `table` function from the module `compute_sum_S_module` and not copy any code from the module file or the program file from Exercise 2.39. Name of program file: `compute_sum_S_cml.py`. ◇

**Exercise 3.19.** *Use options and values in Exer. 3.18.*

Let the input to the program in Exercise 3.18 be option-value pairs of the type `-t`, `-T`, and `-n`, with sensible default values for these quantities set in the program. Apply the `getopt` module to read the command-line arguments. Name of program file: `compute_sum_S_cml_getopt.py`. ◇

**Exercise 3.20.** *Use optparse in the program from Ch. 3.2.4.*

Python has a module `optparse`, which is an alternative to `getopt` for reading `-option value` pairs. Read about `optparse` in the official Python documentation, either the Python Library Reference or the Global Module Index. Figure out how to apply `optparse` to the `location.py` program from Chapter 3.2.4 and modify the program to make use of `optparse` instead of `getopt`. Name of program file: `location_optparse.py`. ◇

**Exercise 3.21.** *Compute the distance it takes to stop a car.*

A car driver, driving at velocity $v_0$, suddenly puts on the brake. What braking distance $d$ is needed to stop the car? One can derive,

from basic physics, that

$$d = \frac{1}{2}\frac{v_0^2}{\mu g}\,. \tag{3.7}$$

Make a program for computing $d$ in (3.7) when the initial car velocity $v_0$ and the friction coefficient $\mu$ are given on the command line. Run the program for two cases: $v_0 = 120$ and $v_0 = 50$ km/h, both with $\mu = 0.3$ ($\mu$ is dimensionless). (Remember to convert the velocity from km/h to m/s before inserting the value in the formula!) Name of program file: `stopping_length.py`. ◇

**Exercise 3.22.** *Check if mathematical rules hold on a computer.*

Because of round-off errors, it could happen that a mathematical rule like $(ab)^3 = a^3b^3$ does not hold (exactly) on a computer. The idea of this exercise is to check such rules for a large number of random numbers. We can make random numbers using the `random` module in Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Here, `a` and `b` will be random numbers which are always larger than or equal to `A` and smaller than `B`.

Make a program that reads the number of tests to be performed from the command line. Set `A` and `B` to fixed values (say -100 and 100). Perform the test in a loop. Inside the loop, draw random numbers `a` and `b` and test if the two mathematical expressions `(a*b)**3` and `a**3*b**3` are equivalent. Count the number of failures of equivalence and write out the percentage of failures at the end of the program.

Duplicate the code segment outlined above to also compare the expressions $a/b$ and $1/(b/a)$. Name of program file: `math_rules_failures.py`. ◇

**Exercise 3.23.** *Improve input to the program in Exer. 3.22.*

The purpose of this exercise is to extend the program from Exercise 3.22 to handle a large number of mathematical rules. Make a function `equal(expr1, expr2, A, B, n=500)` which tests if the mathematical expressions `expr1` and `expr2`, given as strings and involving numbers `a` and `b`, are exactly equal (`eval(expr1) == eval(expr2)`) for `n` random choices of numbers `a` and `b` in the interval between `A` and `B`. Return the percentage of failures. Make a module with the `equal` function and a test block which feeds the `equal` function with arguments read from the command line. Run the module file as a program to test the two rules from Exercise 3.22. Also test the rules $e^{a+b} = a^a e^b$ and $\ln a^b = b \ln a$ (take a `from math import *` in the module file so that mathematical functions like `exp` and `log` are defined). Name of program file: `math_rules_failures_cml.py`. ◇

**Exercise 3.24.** *Apply the program from Exer. 3.23.*

Import the `equal` function from the module made in Exercise 3.23 and test the three rules from Exercise 3.22 in addition to the following rules:

- $a - b$ and $-(b - a)$
- $a/b$ and $1/(b/a)$
- $(ab)^4$ and $a^4 b^4$
- $(a + b)^2$ and $a^2 + 2ab + b^2$
- $(a + b)(a - b)$ and $a^2 - b^2$
- $e^{a+b}$ and $e^a e^b$
- $\ln a^b$ and $b \ln a$
- $\ln ab$ and $\ln a + \ln b$
- $ab$ and $e^{\ln a + \ln b}$
- $1/(1/a + 1/b)$ and $ab/(a + b)$
- $a(\sin^2 b + \cos^2 b)$ and $a$
- $\sinh(a + b)$ and $(e^a e^b - e^{-a} e^{-b})/2$
- $\tan(a + b)$ and $\sin(a + b)/\cos(a + b)$
- $\sin(a + b)$ and $\sin a \cos b + \sin b \cos a$

Store all the expressions in a list of 2-tuples, where each 2-tuple contains two mathematically equivalent expressions as strings which can be sent to the `eval` function. Choose `A` as 1 and `B` as 50. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate corresponding to the `B` values. Does the failure rate depend on the magnitude of the numbers $a$ and $b$? Name of program file: `math_rules_failures_table.py`.

*Remark.* Exercise 3.22 can be solved by a simple program, but if you want to check 17 rules the present exercise demonstrates how important it is to be able to automate the process via the `equal` function and two nested loops over a list of equivalent expressions. ◇

**Exercise 3.25.** *Compute the binomial distribution.*

Consider an uncertain event where there are two outcomes only, typically success or failure. Flipping a coin is an example: The outcome is uncertain and of two types, either head (can be considered as success) or tail (failure). Throwing a die can be another example, if (e.g.) getting a six is considered success and all other outcomes represent failure. Let the probability of success be $p$ and that of failure $1-p$. If we perform $n$ experiments, where the outcome of each experiment does not depend on the outcome of previous experiments, the probability of getting success $x$ times (and failure $n - x$ times) is given by

$$B(x, n, p) = \frac{n!}{x!(n - x)!} p^x (1 - p)^{n-x} . \tag{3.8}$$

This formula (3.8) is called the binomial distribution. The expression $x!$ is the factorial of $x$ as defined in Exercise 2.33. Implement (3.8) in a function `binomial(x, n, p)`. Make a module containing this `binomial` function. Include a test block at the end of the module file. Name of program file: `binomial_distribution.py`. ◇

**Exercise 3.26.** *Apply the binomial distribution.*

Use the module from Exercise 3.25 to make a program for solving the problems below.

1. What is the probability of getting two heads when flipping a coin five times?
   This probability corresponds to $n = 5$ events, where the success of an event means getting head, which has probability $p = 1/2$, and we look for $x = 2$ successes.
2. What is the probability of getting four ones in a row when throwing a die?
   This probability corresponds to $n = 4$ events, success is getting one and has probability $p = 1/6$, and we look for $x = 4$ successful events.
3. Suppose cross country skiers typically experience one ski break in one out of 120 competitions. Hence, the probability of breaking a ski can be set to $p = 1/120$. What is the probability $b$ that a skier will experience a ski break during five competitions in a world championship?
   This question is a bit more demanding than the other two. We are looking for the probability of 1, 2, 3, 4 or 5 ski breaks, so it is simpler to ask for the probability $c$ of *not* breaking a ski, and then compute $b = 1 - c$. Define "success" as breaking a ski. We then look for $x = 0$ successes out of $n = 5$ trials, with $p = 1/120$ for each trial. Compute $b$.

Name of program file: `binomial_problems.py`. ◇

**Exercise 3.27.** *Compute probabilities with the Poisson distribution.*

Suppose that over a period of $t_m$ time units, a particular uncertain event happens (on average) $\nu t_m$ times. The probability that there will be $x$ such events in a time period $t$ is approximately given by the formula

$$P(x, t, \nu) = \frac{(\nu t)^x}{x!} e^{-\nu t} . \tag{3.9}$$

This formula is known as the Poisson distribution[12]. An important assumption is that all events are independent of each other and that the probability of experiencing an event does not change significantly over time.

[12] It can be shown that (3.9) arises from (3.8) when the probability $p$ of experiencing the event in a small time interval $t/n$ is $p = \nu t/n$ and we let $n \to \infty$.

Implement (3.9) in a function `Poisson(x, t, nu)`, and make a program that reads $x$, $t$, and $\nu$ from the command line and writes out the probability $P(x, t, \nu)$. Use this program to solve the problems below.

1. Suppose you are waiting for a taxi in a certain street at night. On average, 5 taxis pass this street every hour at this time of the night. What is the probability of not getting a taxi after having waited 30 minutes?
   Since we have 5 events in a time period of $t_m = 1$ hour, $\nu t_m = \nu = 5$. The sought probability is then $P(0, 1/2, 5)$. Compute this number. What is the probability of having to wait two hours for a taxi?
   If 8 people need two taxis, that is the probability that two taxis arrive in a period of 20 minutes?
2. In a certain location, 10 earthquakes have been recorded during the last 50 years. What is the probability of experiencing exactly three earthquakes over a period of 10 years in this erea? What is the probability that a visitor for one week does not experience any earthquake?
   With 10 events over 50 years we have $\nu t_m = \nu \cdot 50$ years $= 10$ events, which imples $\nu = 1/5$ event per year. The answer to the first question of having $x = 3$ events in a period of $t = 10$ years is given directly by (3.9). The second question asks for $x = 0$ events in a time period of 1 week, i.e., $t = 1/52$ years, so the answer is $P(0, 1/52, 1/5)$.
3. Suppose that you count the number of misprints in the first versions of the reports you write and that this number shows an average of six misprints per page. What is the probability that a reader of a first draft of one of your reports reads six pages without hitting a misprint?
   Assuming that the Poisson distribution can be applied to this problem, we have "time" $t_m$ as 1 page and $\nu \cdot 1 = 6$, i.e., $\nu = 6$ events (misprints) per page. The probability of no events in a "period" of six pages is $P(0, 6, 6)$.

◇

# Array Computing and Curve Plotting 4

Lists are introduced in Chapter 2 to store "tabular data" in a convenient way. An array is an object that is very similar to a list, but less flexible and computationally much more efficient. When using the computer to perform mathematical calculations, we often end up with a huge amount of numbers and associated arithmetic operations. Storing numbers in lists may in such contexts lead to slow programs, while arrays can make the programs run much faster. This may not be very important for the mathematical problems in this book, since most of the programs usually finish execution within a few seconds. Nevertheless, in more advanced applications of mathematics, especially the applications met in industry and science, computer programs may run for weeks and months. Any clever idea that reduces the execution time to days or hours is therefore paramount[1].

This chapter gives a brief introduction to arrays – how they are created and what they can be used for. Array computing usually ends up with a lot of numbers. It may be very hard to understand what these numbers mean by just looking at them. Since the human is a visual animal, a good way to understand numbers is to visualize them. In this chapter we concentrate on visualizing curves that reflect functions of one variable, e.g., curves of the form $y = f(x)$. A synonym for curve is graph, and the image of curves on the screen is often called a plot. We will use arrays to store the information about points along the curve. It is fair to say that array computing demands visualization and visualization demands arrays.

[1] Many may argue that programmers of mathematical software have traditionally paid too much attention to efficiency and smart program constructs. The resulting software often becomes very hard to maintain and extend. In this book we advocate a focus on clear, well-designed, and easy-to-understand programs that work correctly. Optimization for speed should always come as a second step in program development.

All program examples in this chapter can be found as files in the folder `src/plot`.

## 4.1 Vectors

This section gives a brief introduction to the vector concept, assuming that you have heard about vectors in the plane and maybe vectors in space before. This background will be valuable when we start to work with arrays and curve plotting.

### 4.1.1 The Vector Concept

Some mathematical quantities are associated with a set of numbers. One example is a point in the plane, where we need two coordinates (real numbers) to describe the point mathematically. Naming the two coordinates of a particular point as $x$ and $y$, it is common to use the notation $(x, y)$ for the point. That is, we group the numbers inside parentheses. Instead of symbols we might use the numbers directly: $(0, 0)$ and $(1.5, -2.35)$ are also examples of coordinates in the plane.

A point in three-dimensional space has three coordinates, which we may name $x_1$, $x_2$, and $x_3$. The common notation groups the numbers inside parentheses: $(x_1, x_2, x_3)$. Alternatively, we may use the symbols $x$, $y$, and $z$, and write the point as $(x, y, z)$, or numbers can be used instead of symbols.

From high school you may have a memory of solving two equations with two unknowns. At the university you will soon meet problems that are formulated as $n$ equations with $n$ unknowns. The solution of such problems contains $n$ numbers that we can collect inside parentheses and number from 1 to $n$: $(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$.

Quantities such as $(x, y)$, $(x, y, z)$, or $(x_1, \ldots, x_n)$ are known as *vectors* in mathematics. A visual representation of a vector is an arrow that goes from the origin to a point. For example, the vector $(x, y)$ is an arrow that goes from $(0, 0)$ to the point with coordinates $(x, y)$ in the plane. Similarly, $(x, y, z)$ is an arrow from $(0, 0, 0)$ to the point $(x, y, z)$ in three-dimensional space.

Mathematicians found it convenient to introduce spaces with higher dimension than three, because when we have a solution of $n$ equations collected in a vector $(x_1, \ldots, x_n)$, we may think of this vector as a point in a space with dimension $n$, or equivalently, an arrow that goes from the origin $(0, \ldots, 0)$ in $n$-dimensional space to the point $(x_1, \ldots, x_n)$. Figure 4.1 illustrates a vector as an arrow, either starting at the origin, or at any other point. Two arrows/vectors that have the same direction and the same length are mathematically equivalent.

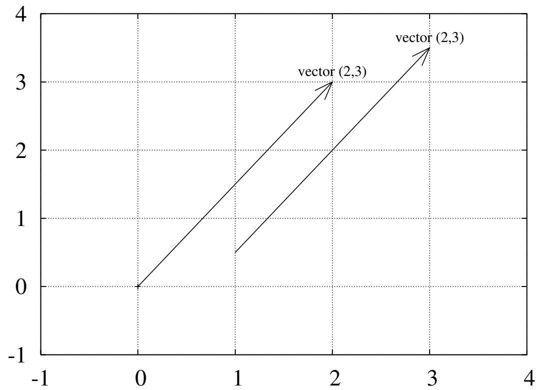



**Fig. 4.1** A vector $(2,3)$ visualized in the standard way as an arrow from the origin to the point $(2,3)$, and mathematically equivalently, as an arrow from $(1,\frac{1}{2})$ (or any point $(a,b)$) to $(3,3\frac{1}{2})$ (or $(a+2,b+3)$).

We say that $(x_1,\ldots,x_n)$ is an $n$-vector or a vector with $n$ components. Each of the numbers $x_1$, $x_2$, ... is a component or an element. We refer to the first component (or element), the second component (or element), and so forth.

A Python program may use a list or tuple to represent a vector:

```
v1 = [x, y]          # list of variables
v2 = (-1, 2)         # tuple of numbers
v3 = (x1, x2, x3)    # tuple of variables
from math import exp
v4 = [exp(-i*0.1) for i in range(150)]
```

While `v1` and `v2` are vectors in the plane and `v3` is a vector in three-dimensional space, `v4` is a vector in a 150-dimensional space, consisting of 150 values of the exponentional function. Since Python lists and tuples have 0 as the first index, we may also in mathematics write the vector $(x_1,x_2)$ as $(x_0,x_1)$. This is not at all common in mathematics, but makes the distance from a mathematical description of a problem to its solution in Python shorter.

It is impossible to visually demonstrate how a space with 150 dimensions looks like. Going from the plane to three-dimensional space gives a rough feeling of what it means to add a dimension, but if we forget about the idea of a visual perception of space, the mathematics is very simple: Going from a 4-dimensional vector to a 5-dimensional vector is just as easy as adding an element to a list of symbols or numbers.

### 4.1.2 Mathematical Operations on Vectors

Since vectors can be viewed as arrows having a length and a direction, vectors are extremely useful in geometry and physics. The velocity of a car has a magnitude and a direction, so has the acceleration, and

the position of a car is a point[2] which is also a vector. An edge of a triangle can be viewed as a line (arrow) with a direction and length.

In geometric and physical applications of vectors, mathematical operations on vectors are important. We shall exemplify some of the most important operations on vectors below. The goal is not to teach computations with vectors, but more to illustrate that such computations are defined by mathematical rules[3]. Given two vectors, $(u_1, u_2)$ and $(v_1, v_2)$, we can add these vectors according to the rule:

$$(u_1, u_2) + (v_1, v_2) = (u_1 + v_1, u_2 + v_2)\,. \tag{4.1}$$

We can also subtract two vectors using a similar rule:

$$(u_1, u_2) - (v_1, v_2) = (u_1 - v_1, u_2 - v_2)\,. \tag{4.2}$$

A vector can be multiplied by a number. This number, called $a$ below, is usually denoted as a *scalar*:

$$a \cdot (v_1, v_2) = (av_1, av_2)\,. \tag{4.3}$$

The inner product, also called dot product, or scalar product, of two vectors is a number[4]:

$$(u_1, u_2) \cdot (v_1, v_2) = u_1 v_1 + u_2 v_2\,. \tag{4.4}$$

There is also a *cross product* defined for 2-vectors or 3-vectors, but we do not list the cross product formula here.
The length of a vector is defined by

$$||(v_1, v_2)|| = \sqrt{(v_1, v_2) \cdot (v_1, v_2)} = \sqrt{v_1^2 + v_2^2}\,. \tag{4.5}$$

The same mathematical operations apply to $n$-dimensional vectors as well. Instead of counting indices from 1, as we usually do in mathematics, we now count from 0, as in Python. The addition and subtraction of two vectors with $n$ components (or elements) read

$$(u_0, \ldots, u_{n-1}) + (v_0, \ldots, v_{n-1}) = (u_0 + v_0, \ldots, u_{n-1} + v_{n-1}), \tag{4.6}$$
$$(u_0, \ldots, u_{n-1}) - (v_0, \ldots, v_{n-1}) = (u_0 - v_0, \ldots, u_{n-1} - v_{n-1})\,. \tag{4.7}$$

[2] A car is of course not a mathematical point, but when studying the acceleration of a car, it suffices to view it as a point. In other occasions, e.g., when simulating a car crash on a computer, the car may be modeled by a large number (say $10^6$) of connected points.

[3] You might recall many of the formulas here from high school mathematics or physics. The really new thing in this chapter is that we show how rules for vectors in the plane and in space can easily be extended to vectors in $n$-dimensional space.

[4] From high school mathematics and physics you might recall that the inner or dot product also can be expressed as the product of the lengths of the two vectors multiplied by the cosine of the angle between them. We will not make use of this formula.

Multiplication of a scalar $a$ and a vector $(v_0, \ldots, v_{n-1})$ equals

$$(av_0, \ldots, av_{n-1})\,. \tag{4.8}$$

The inner or dot product of two $n$-vectors is defined as

$$(u_0, \ldots, u_{n-1}) \cdot (v_0, \ldots, v_{n-1}) = u_0 v_0 + \cdots + u_{n-1} v_{n-1} = \sum_{j=0}^{n-1} u_j v_j\,. \tag{4.9}$$

Finally, the length $||v||$ of an $n$-vector $v = (v_0, \ldots, v_{n-1})$ is

$$\begin{aligned} \sqrt{(v_0, \ldots, v_{n-1}) \cdot (v_0, \ldots, v_{n-1})} &= \left(v_0^2 + v_1^2 + \cdots + v_{n-1}^2\right)^{\frac{1}{2}} \\ &= \left(\sum_{j=0}^{n-1} v_j^2\right)^{\frac{1}{2}}\,. \end{aligned} \tag{4.10}$$

### 4.1.3 Vector Arithmetics and Vector Functions

In addition to the operations on vectors in Chapter 4.1.2, which you might recall from high school mathematics, we can define other operations on vectors. This is very useful for speeding up programs. Unfortunately, the forthcoming vector operations are hardly treated in textbooks on mathematics, yet these operations play a significant role in mathematical software, especially in computing environment such as Matlab, Octave, Python, and R.

Applying a mathematical function of one variable, $f(x)$, to a vector is defined as a vector where $f$ is applied to each element. Let $v = (v_0, \ldots, v_{n-1})$ be a vector. Then

$$f(v) = (f(v_0), \ldots, f(v_{n-1}))\,.$$

For example, the sine of $v$ is

$$\sin(v) = (\sin(v_0), \ldots, \sin(v_{n-1}))\,.$$

It follows that squaring a vector, or the more general operation of raising the vector to a power, can be defined as applying the operation to each element:

$$v^b = (v_0^b, \ldots, v_{n-1}^b)\,.$$

Another operation between two vectors that arises in computer programming of mathematics is the "asterix" multiplication, defined as

$$u * v = (u_0 v_0, u_1 v_1, \ldots, u_{n-1} v_{n-1})\,. \tag{4.11}$$

Adding a scalar to a vector or array can be defined as adding the scalar to each component. If $a$ is a scalar and $v$ a vector, we have

$$a + v = (a + v_0, \ldots, a + v_{n-1}) .$$

A compound vector expression may look like

$$v^2 * \cos(v) * e^v + 2 . \tag{4.12}$$

How do we calculate this expression? We use the normal rules of mathematics, working our way, term by term, from left to right, paying attention to the fact that powers are evaluated before multiplications and divisions, which are evaluated prior to addition and subtraction. First we calculate $v^2$, which results in a vector we may call $u$. Then we calculate $\cos(v)$ and call the result $p$. Then we multiply $u * p$ to get a vector which we may call $w$. The next step is to evaluate $e^v$, call the result $q$, followed by the multiplication $w*q$, whose result is stored as $r$. Then we add $r + 2$ to get the final result. It might be more convenient to list these operations after each other:

1. $u = v^2$
2. $p = \cos(v)$
3. $w = u * p$
4. $q = e^v$
5. $r = w * q$
6. $s = r + 2$

Writing out the vectors $u$, $w$, $p$, $q$, and $r$ in terms of a general vector $v = (v_0, \ldots, v_{n-1})$ (do it!) shows that the result of the expression (4.12) is the vector

$$(v_0^2 \cos(v_0)e^{v_0} + 2, \ldots, v_{n-1}^2 \cos(v_{n-1})e^{v_{n-1}} + 2) .$$

That is, component no. $i$ in the result vector equals the number arising from applying the formula (4.12) to $v_i$, where the * multiplication is ordinary multiplication between two numbers.

We can, alternatively, introduce the function

$$f(x) = x^2 \cos(x)e^x + 2$$

and use the result that $f(v)$ means applying $f$ to each element in $v$. The result is the same as in the vector expression (4.12).

In Python programming it is important for speed (and convenience too) that we can apply functions of one variable, like $f(x)$, to vectors. What this means mathematically is something we have tried to explain in this subsection. Doing Exercises 4.4 and 4.5 may help to grasp the ideas of vector computing, and with more programming experience you will hopefully discover that vector computing is very useful. It is not necessary to have a thorough understanding of vector computing in order to proceed with the next sections.

Arrays are used to represent vectors in a program, but one can do more with arrays than with vectors. Until Chapter 4.6 it suffices to think of arrays as the same as vectors in a program.

## 4.2 Arrays in Python Programs

This section introduces array programming in Python, but first we create some lists and show how arrays differ from lists.

### 4.2.1 Using Lists for Collecting Function Data

Suppose we have a function $f(x)$ and want to evaluate this function at a number of $x$ points $x_0, x_1, \ldots, x_{n-1}$. We could collect the $n$ pairs $(x_i, f(x_i))$ in a list, or we could collect all the $x_i$ values, for $i = 0, \ldots, n-1$, in a list and all the associated $f(x_i)$ values in another list. We learned how to create such lists in Chapter 2, but as a review, we present the relevant program statements in an interactive session:

```
>>> def f(x):
...     return x**3       # sample function
...
>>> n = 5                 # no of points along the x axis
>>> dx = 1.0/(n-1)        # spacing between x points in [0,1]
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here we have used list comprehensions for achieving compact code. Make sure that you understand what is going on in these list comprehensions (you are encouraged to write the same code using standard `for` loops and appending new list elements in each pass of the loops).

The list elements consist of objects of the same type: any element in `pairs` is a list of two `float` objects, while any element in `xlist` or `ylist` is a `float`. Lists are more flexible than that, because an element can be an object of any type, e.g.,

```
mylist = [2, 6.0, 'tmp.ps', [0,1]]
```

Here `mylist` holds an `int`, a `float`, a string, and a list. This combination of diverse object types makes up what is known as *heterogeneous* lists. We can also easily remove elements from a list or add new elements anywhere in the list. This flexibility of lists is in general convenient to have as a programmer, but in cases where the elements are of the same type and the number of elements is fixed, arrays can be used instead. The benefits of arrays are faster computations, less memory demands, and extensive support for mathematical operations on the

data. Because of greater efficiency and mathematical convenience, arrays will be used to a large extent in this book. The great use of arrays is also prominent in other programming environments such as Matlab, Octave, and R, for instance. Lists will be our choice instead of arrays when we need the flexibility of adding or removing elements or when the elements may be of different object types.

### 4.2.2 Basics of Numerical Python Arrays

An *array* object can be viewed as a variant of a list, but with the following assumptions and features:

- All elements must be of the same type, preferably integer, real, or complex numbers, for efficient numerical computing and storage.
- The number of elements must be known[5] when the array is created.
- Arrays are not part of standard Python[6] – one needs an additional package called *Numerical Python*, often abbreviated as NumPy. The Python name of the package, to be used in `import` statements, is `numpy`.
- With `numpy`, a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called *vectorization* and may cause a dramatic speed-up of Python programs. Vectorization makes use of the vector computing concepts from Chapter 4.1.3.
- Arrays with one index are often called vectors. Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists. Arrays can also have three or more indices.

The fundamental import statement to get access to Numerical Python array functionality reads

```
from numpy import *
```

To convert a list `r` to an array, we use the `array` function from `numpy`:

```
a = array(r)
```

To create a new array of length `n`, filled with zeros, we write

```
a = zeros(n)
```

The array elements are of a type that corresponds to Python's `float` type. A second argument to `zeros` can be used to specify other element types, e.g., `int`. Arrays with more than one index are treated in Chapter 4.6.

[5] The number of elements can be changed, at a substantial computational cost.

[6] Actually, there is an object type called `array` in standard Python, but this data type is not so efficient for mathematical computations.

Often one wants an array to have $n$ elements with uniformly distributed values in an interval $[p, q]$. The `numpy` function `linspace` creates such arrays:

```
a = linspace(p, q, n)
```

We remark that there are a large number of functions and modules within `numpy`.

Array elements are accessed by square brackets as for lists: `a[i]`. Slices also work as for lists, for example, `a[1:-1]` picks out all elements except the first and the last, but contrary to lists, `a[1:-1]` is not a copy of the data in `a`. Hence,

```
b = a[1:-1]
b[2] = 0.1
```

will also change `a[3]` to `0.1`. A slice `a[i:j:s]` picks out the elements starting with index `i` and stepping `s` indices at the time up to, but not including, `j`. Omitting `i` implies `i=0`, and omitting `j` implies `j=n` if `n` is the number of elements in the array. For example, `a[0:-1:2]` picks out every two elements up to, but not including, the last element, while `a[::4]` picks out every four elements in the whole array.

### 4.2.3 Computing Coordinates and Function Values

With these basic operations at hand, we can continue the session from the previous section and make arrays out of the lists `xlist`, `ylist`, and `pairs`:

```
>>> from numpy import *
>>> x2 = array(xlist)      # turn list xlist into array x2
>>> y2 = array(ylist)
>>> x2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

Instead of first making a list and then converting the list to an array, we can compute the arrays directly. The equally spaced coordinates in `x2` are naturally computed by the `linspace` function. The `y2` array can be created by `zeros`, to ensure that `y2` has the right length[7] `len(x2)`, and then we can run a `for` loop to fill in all elements in `y2` with `f` values:

```
>>> n = len(xlist)
>>> x2 = linspace(0, 1, n)
>>> y2 = zeros(n)
>>> for i in xrange(n):
```

[7] This is referred to as *allocating* the array, and means that a part of the computer's memory is marked for being occupied by this array.

```
...     y2[i] = f(x2[i])
...
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

Note that we here in the `for` loop have used `xrange` instead of `range`. The former is faster for long loops and is our preference over `range` when we have loops over (possibly long) arrays.

We used a list comprehension for computing the `y`, while we used a `for` loop for computing the array `y2`. List comprehensions do not work with arrays because the list comprehension creates a list, not an array. We can, of course, compute the $y$ coordinates with a list comprehension and then turn the resulting list into an array:

```
>>> x2 = linspace(0, 1, n)
>>> y2 = array([f(xi) for xi in x2])
```

Nevertheless, there is a better way of computing `y2` as the next paragraph explains.

### 4.2.4 Vectorization

Loops over very long arrays may run slowly. A great advantage with arrays is that we can get rid of the loops and apply `f` directly to the whole array:

```
>>> y2 = f(x2)
>>> y2
array([ 0.      ,  0.015625,  0.125   ,  0.421875,  1.      ])
```

The magic that makes `f(x2)` work builds on the vector computing concepts from Chapter 4.1.3.

Instead of calling `f(x2)` we can equivalently write the function formula `x2**3` directly. As another example, a Python assignment like

```
r = sin(x)*cos(x)*exp(-x**2) + 2 + x**2
```

works perfectly for an array `x`. The resulting array is the same as if we apply the formula to each array entry:

```
r = zeros(len(x))
for i in xrange(len(x)):
    r[i] = sin(x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

Replacing a loop like the one above by a vector/array expression (like `sin(x)*cos(x)*exp(-x**2) + 2 + x**2`) is what we call vectorization. The loop version is often referred to as *scalar code*. For example,

```
x = zeros(N);  y = zeros(N)
dx = 2.0/(N-1) # spacing of x coordinates
for i in range(N):
    x[i] = -1 + dx*i
    y[i] = exp(-x[i])*x[i]
```

is scalar code, while the corresponding vectorized version reads

```
x = linspace(-1, 1, N)
y = exp(-x)*x
```

We remark that list comprehensions,

```
x = array([-1 + dx*i for i in range(N)])
y = array([exp(-xi)*xi for xi in x])
```

result in scalar code because we still have explicit, slow Python `for` loops. The requirement of vectorized code is that there are no explicit Python `for` loops. The loops that are required to compute each array element are performed in fast C or Fortran code in the `numpy` package.

Most Python functions intended for an scalar argument `x`, like

```
def f(x):
    return x**4*exp(-x)
```

automatically work for an array argument `x`:

```
x = linspace(-3, 3, 101)
y = f(x)
```

We say that the function `f` is vectorized. Not any Python function `f(x)` will be automatically vectorized, i.e., sending an array `x` to `f(x)` may lead to wrong results or an exception. Chapter 4.4.1 provides examples where we have to do special actions in order to vectorize functions.

Vectorization is very important for speeding up Python programs where we do heavy computations with arrays. Moreover, vectorization gives more compact code that is easier to read. Vectorization becomes particularly important for statistical simulations in Chapter 8.

## 4.3 Curve Plotting

Visualizing a function $f(x)$ is done by drawing the curve $y = f(x)$ in an $xy$ coordinate system. When we use a computer to do this task, we say that we *plot* the curve. Technically, we plot a curve by drawing straight lines between $n$ points on the curve. The more points we use, the smoother the curve appears.

Suppose we want to plot the function $f(x)$ for $a \leq x \leq b$. First we pick out $n$ $x$ coordinates in the interval $[a, b]$, say we name these $x_0, x_1, \ldots, x_{n-1}$. Then we evaluate $y_i = f(x_i)$ for $i = 0, 1, \ldots, n-1$.

The points $(x_i, y_i)$, $i = 0, 1, \ldots, n-1$, now lie on the curve $y = f(x)$. Normally, we choose the $x_i$ coordinates to be equally spaced, i.e.,

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}\,.$$

If we store the $x_i$ and $y_i$ values in two arrays `x` and `y`, we can plot the curve by the command `plot(x,y)`.

Sometimes the names of the independent variable and the function differ from $x$ and $f$, but the plotting procedure is the same. Our first example of curve plotting demonstrates this fact by involving a function of $t$.

### 4.3.1 The SciTools and Easyviz Packages

There are lots of plotting programs that we can use to create visual graphics with curves on the computer screen. As part of this book project, we have created a unified interface *Easyviz* to different plotting programs such that you can write one set of statements in your program regardless of which plotting program you actually use "behind the curtain" to create the graphics. The statements needed to plot a curve are very similar to those used in the Matlab and Octave computing environments.

Easyviz is part of a larger package called SciTools. This package contains many useful tools for mathematical computations in Python. SciTools builds heavily on Numerical Python. It also makes use of the comprehensive scientific computing environment SciPy. If you start your program with

```
from scitools.std import *
```

you will automatically perform import of many useful modules for numerical Python programming. Among Easyviz functions and other things, the import statement above performs imports from `numpy`, `scitools.numpyutils` (some convenience functions), `numpy.lib.scimath` (see Chapter 1.6.3), and `scipy` (if available). In addition, the statement imports `os`, `sys`, and the `StringFunction` tool (see Chapter 3.1.4). We refer to the paragraph "Importing Just Easyviz" on page 194 for a precise list of what is actually imported by a `from scitools.std import *`. The advantage with this particular import is that one line of code gives you a lot of the functionality you commonly need in this book. The downside is that this import statement is comprehensive and therefore takes some time (seconds) to execute, especially if `scipy` is available. If you find the waiting time annoying, you may instead use a minimal set of import statements as explained on page 194.

There are a couple of SciTools functions that you may find convenient to know about:

- `seq(a,b,h)` returns an array with equally spaced numbers starting with `a`, ending with `b`, and with a spacing of `h`.
- `iseq(a,b,h)` works as `seq(a,b,h)` except that `a`, `b`, and `h` are integers and the return array contains a set of integers. The advantage of `iseq` over `range` is that the upper limit `b` is included in the sequence of integers. When implementing mathematical algorithms where an index has a specified range, say $i = 1, \ldots, n$, we think it is clearer to write `for i in iseq(1,n)` in the program instead of `for i in range(1,n+1)`.

The inverse trigonometric functions have different names in `math` and `numpy`, a fact that prevents an expression written for scalars, using `math` names, to be immediately valid for vectors. Therefore, the `from scitools.std import *` action also import the names `asin`, `acos`, and `atan` for `numpy`/`scipy`'s `arcsin`, `arccos`, and `arctan` functions, to ease vectorization of mathematical expressions involving inverse trigonometric functions.

### 4.3.2 Plotting a Single Curve

Let us plot the curve $y = t^2 \exp(-t^2)$ for $t$ values between 0 and 3. First we generate equally spaced coordinates for $t$, say 51 values (50 intervals). Then we compute the corresponding $y$ values at these points, before we call the `plot(t,y)` command to make the curve plot. Here is the complete program:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = zeros(len(t))         # allocate y with float elements
for i in xrange(len(t)):
    y[i] = f(t[i])

plot(t, y)
```

The first line imports all of SciTools and Easyviz that can be handy to have when doing scientific computations. In this program we pre-allocate the `y` array and fill it with values, element by element, in a Python loop. Alternatively, we may operate on the whole `t` array at once, which yields faster and shorter code:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = f(t)                  # compute all f values at once
plot(t, y)
```

The `f` function can also be skipped, if desired, so that we can write directly

```
y = t**2*exp(-t**2)
```

To include the plot in electronic documents, we need a hardcopy of the figure in PostScript, PNG, or another image format. The `hardcopy` command produces files with images in various formats:

```
hardcopy('tmp1.eps') # produce PostScript
hardcopy('tmp1.png') # produce PNG
```

The filename extension determines the format: `.ps` or `.eps` for PostScript, and `.png` for PNG. Figure 4.2 displays the resulting plot.

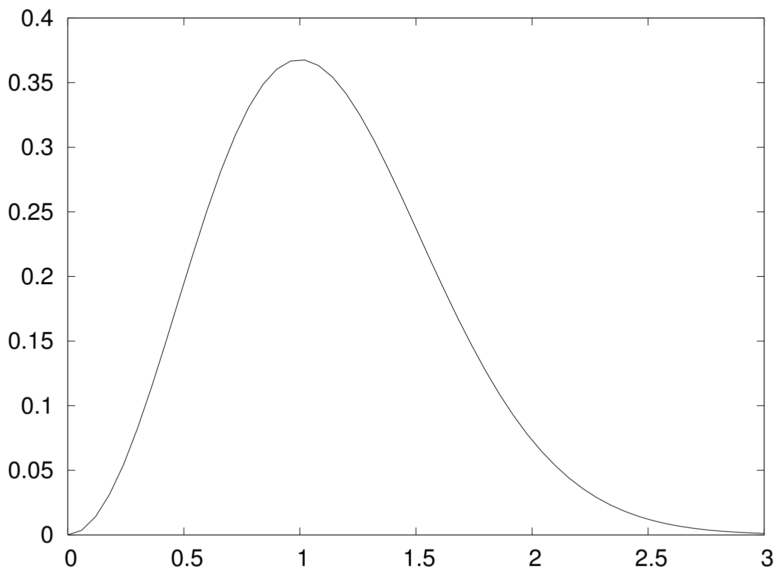



**Fig. 4.2** A simple plot in PostScript format.

On some platforms, some backends may result in a plot that is shown in just a fraction of a second on the screen before the plot window disappears (using the Gnuplot backend on Windows machines or using the Matplotlib backend constitute two examples). To make the window stay on the screen, add

```
raw_input('Press Enter: ')
```

at the end of the program. The plot window is killed when the program terminates, and this satement postpones the termination until the user hits the Enter key.

### 4.3.3 Decorating the Plot

The $x$ and $y$ axis in curve plots should have labels, here $t$ and $y$, respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. In addition, we may want to control the extent of the axes (although most plotting programs will automatically adjust the axes to the range of the data). All such things are easily added after the `plot` command:

```
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')
```

This syntax is inspired by Matlab to make the switch between Easyviz and Matlab almost trivial. Easyviz has also introduced a more "Pythonic" `plot` command where all the plot properties can be set at once:

```
plot(t, y,
     xlabel='t',
     ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     hardcopy='tmp1.eps',
     show=True)
```

With `show=False` one can avoid the plot window on the screen and just make the hardcopy. This feature is particularly useful if one generates a large number of plots in a loop.

Note that we in the curve legend write `t` square as `t^2` (LaTeX style) rather than `t**2` (program style). Whichever form you choose is up to you, but the LaTeX form sometimes looks better in some plotting programs (Gnuplot is one example). See Figure 4.3 for what the modified plot looks like and how `t^2` is typeset in Gnuplot.

### 4.3.4 Plotting Multiple Curves

A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2 \exp(-t^2)$ and $f_2(t) = t^4 \exp(-t^2)$. If we write two `plot` commands after each other, two separate plots will be made. To make the second `plot` command draw the curve in the first plot, we need to issue a `hold('on')` command. Alternatively, we can provide all data in a single `plot` command. A complete program illustrates the different approaches:

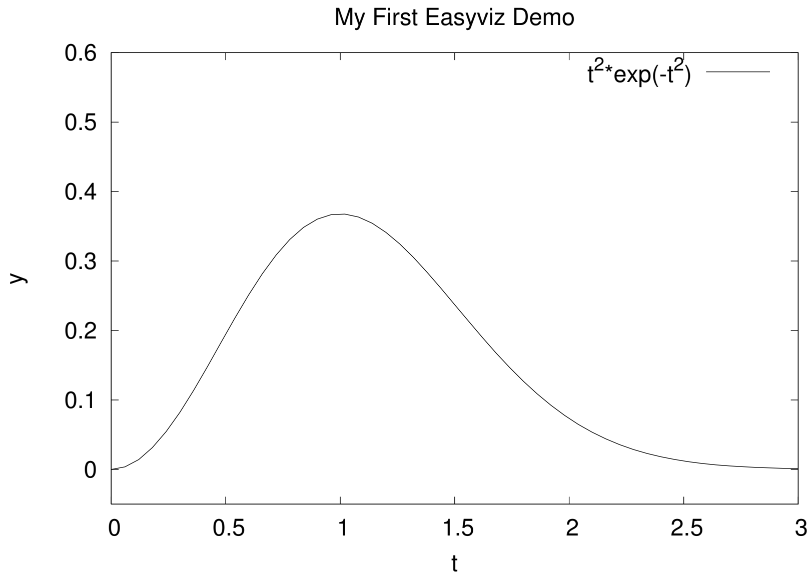



**Fig. 4.3** A single curve with label, title, and axis adjusted.

```
from scitools.std import *   # for curve plotting

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

# Matlab-style syntax:
plot(t, y1)
hold('on')
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')

# alternative:
plot(t, y1, t, y2, xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')
```

The sequence of the multiple legends is such that the first legend corresponds to the first curve, the second legend to the second curve, and so on. The visual result appears in Figure 4.4.

Doing a `hold('off')` makes the next `plot` command create a new plot.

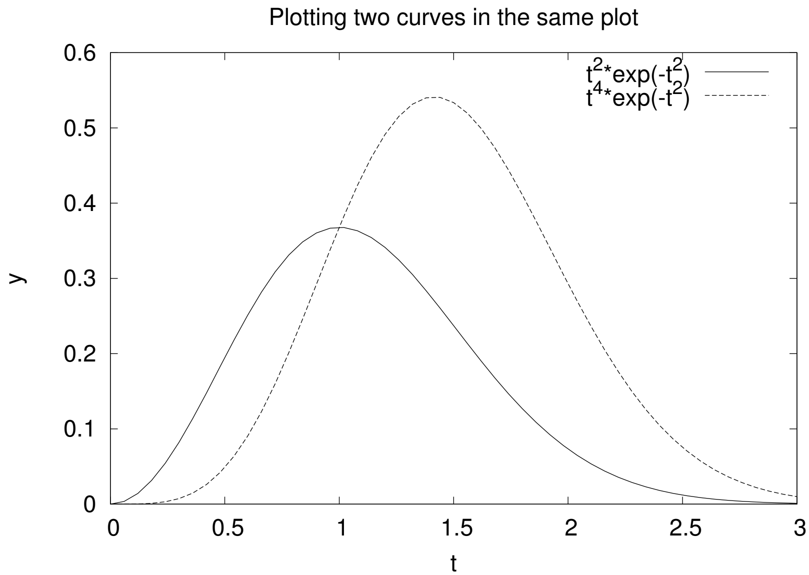



**Fig. 4.4** Two curves in the same plot.

### 4.3.5 Controlling Line Styles

When plotting multiple curves in the same plot, the individual curves get distinct default line styles, depending on the program that is used to produce the curve (and the settings for this program). It might well happen that you get a green and a red curve (which is bad for a significant portion of the male population). Therefore, we often want to control the line style in detail. Say we want the first curve (`t` and `y1`) to be drawn as a red solid line and the second curve (`t` and `y2`) as blue circles at the discrete data points. The Matlab-inspired syntax for specifying line types applies a letter for the color and a symbol from the keyboard for the line type. For example, `r-` represents a red (`r`) line (`-`), while `bo` means blue (`b`) circles (`o`). The line style specification is added as an argument after the $x$ and $y$ coordinate arrays of the curve:

```
plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'bo')

# or
plot(t, y1, 'r-', t, y2, 'bo')
```

The effect of controlling the line styles can be seen in Figure 4.5.

Assume now that we want to plot the blue circles at every 4 points only. We can grab every 4 points out of the `t` array by using an appropriate slice: `t2 = t[::4]`. Note that the first colon means the range from the first to the last data point, while the second colon separates

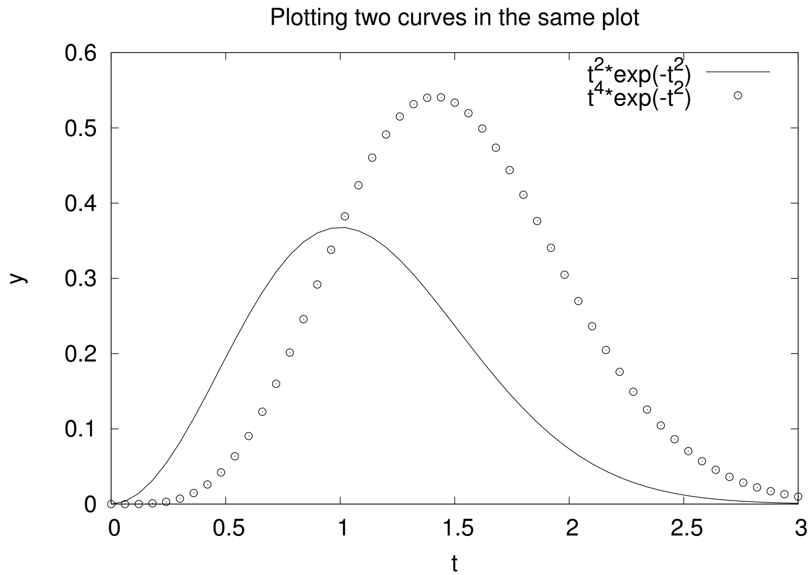



**Fig. 4.5** Two curves in the same plot, with controlled line styles.

this range from the stride, i.e., how many points we should "jump over" when we pick out a set of values of the array.

```
from scitools.std import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
t2 = t[::4]
y2 = f2(t2)

plot(t, y1, 'r-6', t2, y2, 'bo3',
     xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6],
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')
```

In this plot we also adjust the size of the line and the circles by adding an integer: `r-6` means a red line with thickness 6 and `bo5` means red circles with size 5. The effect of the given line thickness and symbol size depends on the underlying plotting program. For the Gnuplot program one can view the effect in Figure 4.6.

The different available line colors include

- yellow: `'y'`
- magenta: `'m'`
- cyan: `'c'`

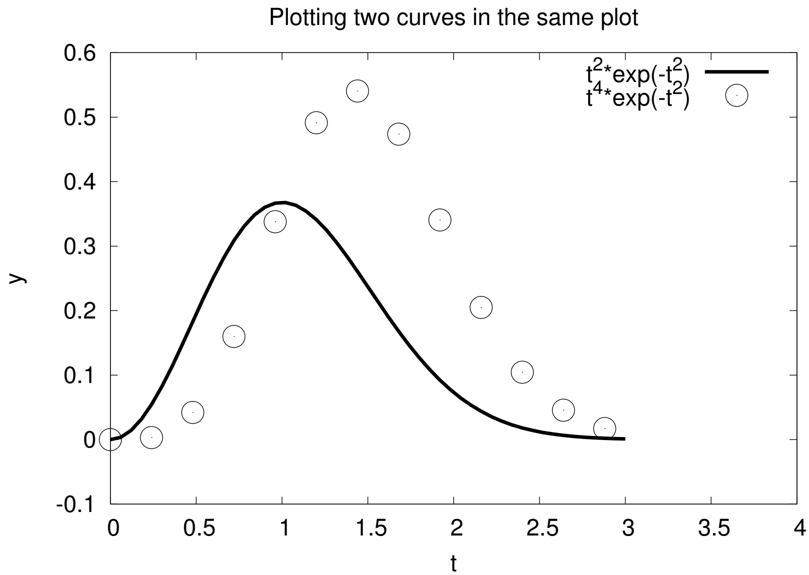



**Fig. 4.6** Circles at every 4 points and extended line thickness (6) and circle size (3).

- red: `'r'`
- green: `'g'`
- blue: `'b'`
- white: `'w'`
- black: `'k'`

The different available line types are

- solid line: `'-'`
- dashed line: `'--'`
- dotted line: `':'`
- dash-dot line: `'-.'`

During programming, you can find all these details in the documentation of the `plot` function. Just type `help(plot)` in an interactive Python shell or invoke `pydoc` with `scitools.easyviz.plot`. This tutorial is available through `pydoc scitools.easyviz`.

We remark that in the Gnuplot program all the different line types are drawn as solid lines on the screen. The hardcopy chooses automatically different line types (solid, dashed, etc.) and not in accordance with the line type specification.

Lots of markers at data points are available:

- plus sign: `'+'`
- circle: `'o'`
- asterisk: `'*'`
- point: `'.'`
- cross: `'x'`

- square: `'s'`
- diamond: `'d'`
- upward-pointing triangle: `'^'`
- downward-pointing triangle: `'v'`
- right-pointing triangle: `'>'`
- left-pointing triangle: `'<'`
- five-point star (pentagram): `'p'`
- six-point star (hexagram): `'h'`
- no marker (default): `None`

Symbols and line styles may be combined, for instance as in `'kx-'`, which means a black solid line with black crosses at the data points.

*Another Example.* Let us extend the previous example with a third curve where the data points are slightly randomly distributed around the $f_2(t)$ curve:

```
from scitools.std import *

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

# pick out each 4 points and add random noise:
t3 = t[::4]      # slice, stride 4
random.seed(11)  # fix random sequence
noise = random.normal(loc=0, scale=0.02, size=len(t3))
y3 = y2[::4] + noise

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'ks-')   # black solid line with squares at data points
plot(t3, y3, 'bo')

legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t')
ylabel('y')
show()
hardcopy('tmp3.eps')
hardcopy('tmp3.png')
```

The plot is shown in Figure 4.7.

*Minimalistic Typing.* When exploring mathematics in the interactive Python shell, most of us are interested in the quickest possible commands. Here is an example of minimalistic syntax for comparing the two sample functions we have used in the previous examples:

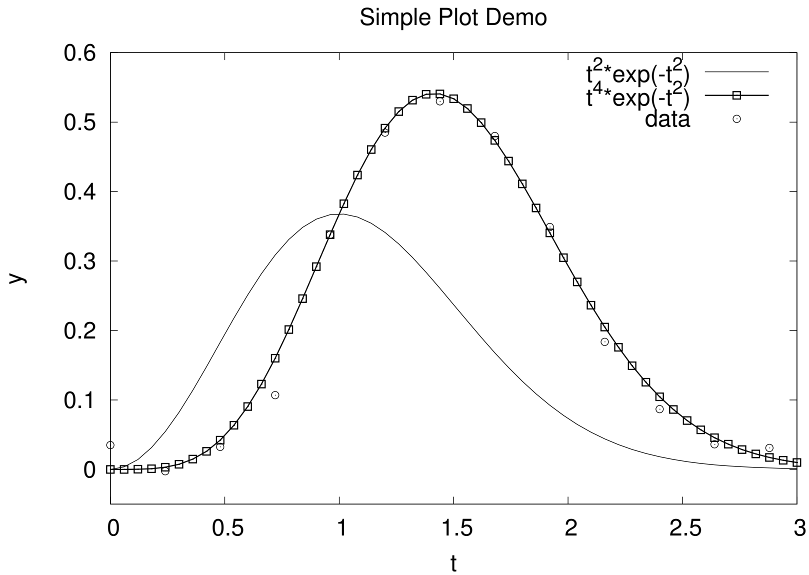



**Fig. 4.7** A plot with three curves.

```
t = linspace(0, 3, 51)
plot(t, t**2*exp(-t**2), t, t**4*exp(-t**2))
```

*Text.* A text can be placed at a point $(x, y)$ using the call

```
text(x, y, 'Some text')
```

*More Examples.* The examples in this tutorial, as well as additional examples, can be found in the `examples` directory in the root directory of the SciTools source code tree.

### 4.3.6 Interactive Plotting Sessions

All the Easyviz commands can of course be issued in an interactive Python session. The only thing to comment is that the `plot` command returns a result:

```
>>> t = linspace(0, 3, 51)
>>> plot(t, t**2*exp(-t**2))
[<scitools.easyviz.common.Line object at 0xb5727f6c>]
```

Most users will just ignore this output line.

All Easyviz commands that produce a plot return an object reflecting the particular type of plot. The `plot` command returns a list of `Line` objects, one for each curve in the plot. These `Line` objects can be invoked to see, for instance, the value of different parameters in the plot:

```
>>> line, = plot(x, y, 'b')
>>> getp(line)
{'description': '',
 'dims': (4, 1, 1),
 'legend': '',
 'linecolor': 'b',
 'pointsize': 1.0,
 ...
```

Such output is mostly of interest to advanced users.

### 4.3.7 Making Animations

A sequence of plots can be combined into an animation and stored in a movie file. First we need to generate a series of hardcopies, i.e., plots stored in files. Thereafter we must use a tool to combine the individual plot files into a movie file.

*Example.* The function $f(x; m, s) = (2\pi)^{-1/2}s^{-1}\exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right]$ is known as the Gaussian function or the probability density function of the normal (or Gaussian) distribution. This bell-shaped function is "wide" for large $s$ and "peak-formed" for small $s$, see Figure 4.8. The function is symmetric around $x = m$ ($m = 0$ in the figure). Our goal is to make an animation where we see how this function evolves as $s$ is decreased. In Python we implement the formula above as a function `f(x, m, s)`.

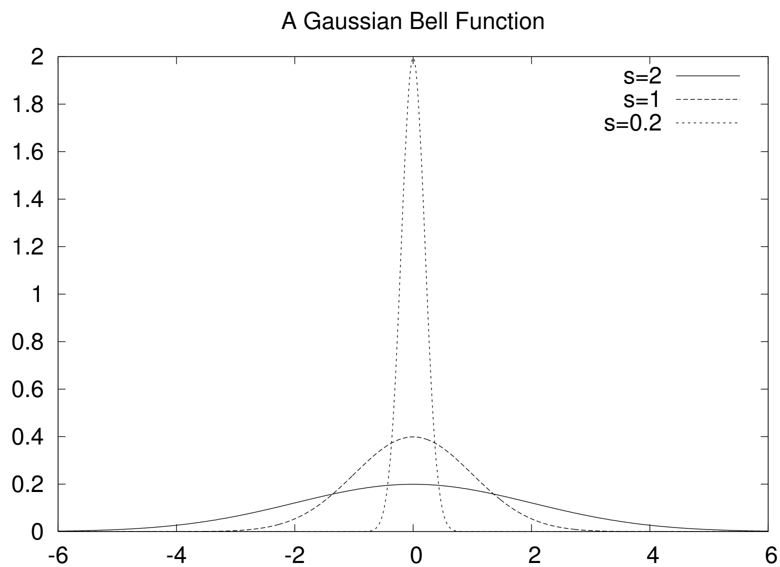



**Fig. 4.8** Different shapes of a Gaussian function.

The animation is created by varying $s$ in a loop and for each $s$ issue a `plot` command. A moving curve is then visible on the screen. One can also make a movie file that can be played as any other computer movie using a standard movie player. To this end, each plot is saved to a file, and all the files are combined together using some suitable tool, which is reached through the `movie` function in Easyviz. All necessary steps will be apparent in the complete program below, but before diving into the code we need to comment upon a couple of issues with setting up the `plot` command for animations.

The underlying plotting program will normally adjust the axis to the maximum and minimum values of the curve if we do not specify the axis ranges explicitly. For an animation such automatic axis adjustment is misleading - the axis ranges must be fixed to avoid a jumping axis. The relevant values for the axis range is the minimum and maximum value of $f$. The minimum value is zero, while the maximum value appears for $x = m$ and increases with decreasing $s$. The range of the $y$ axis must therefore be $[0, f(m; m, \min s)]$.

The function $f$ is defined for all $-\infty < x < \infty$, but the function value is very small already $3s$ away from $x = m$. We may therefore limit the $x$ coordinates to $[m - 3s, m + 3s]$.

Now we are ready to take a look at the complete code for animating how the Gaussian function evolves as the $s$ parameter is decreased from 2 to 0.2:

```
from scitools.std import *
import time

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0
s_start = 2
s_stop = 0.2
s_values = linspace(s_start, s_stop, 30)
x = linspace(m -3*s_start, m + 3*s_start, 1000)
# f is max for x=m; smaller s gives larger max value
max_f = f(m, m, s_stop)

# show the movie on the screen
# and make hardcopies of frames simultaneously:
counter = 0
for s in s_values:
    y = f(x, m, s)
    plot(x, y, axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend='s=%4.2f' % s,
         hardcopy='tmp%04d.png' % counter)
    counter += 1
    #time.sleep(0.2)  # can insert a pause to control movie speed

# make movie file the simplest possible way:
movie('tmp*.png')
```

Note that the $s$ values are decreasing (`linspace` handles this automatically if the start value is greater than the stop value). Also note

that we, simply because we think it is visually more attractive, let the $y$ axis go from -0.1 although the $f$ function is always greater than zero.

*Remarks on Filenames.* For each frame (plot) in the movie we store the plot in a file. The different files need different names and an easy way of referring to the set of files in right order. We therefore suggest to use filenames of the form `tmp0001.png`, `tmp0002.png`, `tmp0003.png`, etc. The printf format `04d` pads the integers with zeros such that `1` becomes `0001`, `13` becomes `0013` and so on. The expression `tmp*.png` will now expand (by an alphabetic sort) to a list of all files in proper order. Without the padding with zeros, i.e., names of the form `tmp1.png`, `tmp2.png`, ..., `tmp12.png`, etc., the alphabetic order will give a wrong sequence of frames in the movie. For instance, `tmp12.png` will appear before `tmp2.png`.

Note that the names of plot files specified when making hardopies must be consistent with the specification of names in the call to `movie`. Typically, one applies a Unix wildcard notation in the call to `movie`, say `plotfile*.eps`, where the asterix will match any set of characters. When specifying hardcopies, we must then use a filename that is consistent with `plotfile*.eps`, that is, the filename must start with `plotfile` and end with `.eps`, but in between these two parts we are free to construct (e.g.) a frame number padded with zeros.

We recommend to always remove previously generated plot files before a new set of files is made. Otherwise, the movie may get old and new files mixed up. The following Python code removes all files of the form `tmp*.png`:

```
import glob, os
for filename in glob.glob('tmp*.png'):
    os.remove(filename)
```

These code lines should be inserted at the beginning of the code example above. Alternatively, one may store all plotfiles in a subfolder and later delete the subfolder. Here is a suitable code segment:

```
import shutil, os
subdir = 'temp'  # subfolder for plot files
if os.path.isdir(subdir):  # does the subfolder already exist?
    shutil.rmtree(subdir)  # delete the whole folder
os.mkdir(subdir) # make new subfolder
os.chdir(subdir) # move to subfolder
# do all the plotting
# make movie
os.chdir(os.pardir)  # optional: move up to parent folder
```

*Movie Formats.* Having a set of (e.g.) `tmp*.png` files, one can simply generate a movie by a `movie('tmp*.png')` call. The `movie` function generates a movie file called `movie.avi` (AVI format), `movie.mpeg` (MPEG format), or `movie.gif` (animated GIF format) in the current working

directory. The movie format depends on the encoders found on your machine.

You can get complete control of the movie format and the name of the movie file by supplying more arguments to the `movie` function. First, let us generate an animated GIF file called `tmpmovie.gif`:

```
movie('tmp_*.eps', encoder='convert', fps=2,
      output_file='tmpmovie.gif')
```

The generation of animated GIF images applies the `convert` program from the ImageMagick suite. This program must of course be installed on the machine. The argument fps stands for frames per second so here the speed of the movie is slow in that there is a delay of half a second between each frame (image file). To view the animated GIF file, one can use the `animate` program (also from ImageMagick) and give the movie file as command-line argument. One can alternatively put the GIF file in a web page in an IMG tag such that a browser automatically displays the movie.

An MPEG movie can be generated by the call

```
movie('tmp_*.eps', encoder='ffmpeg', fps=4,
      output_file='tmpmovie1.mpeg',
```

Alternatively, we may use the `ppmtompeg` encoder from the Netpbm suite of image manipulation tools:

```
movie('tmp_*.eps', encoder='ppmtompeg', fps=24,
      output_file='tmpmovie2.mpeg',
```

The `ppmtompeg` supports only a few (high) frame rates.

The next sample call to `movie` uses the Mencoder tool and specifies some additional arguments (video codec, video bitrate, and the quantization scale):

```
movie('tmp_*.eps', encoder='mencoder', fps=24,
      output_file='tmpmovie.mpeg',
      vcodec='mpeg2video', vbitrate=2400, qscale=4)
```

Playing movie files can be done by a lot of programs. Windows Media Player is a default choice on Windows machines. On Unix, a variety of tools can be used. For animated GIF files the `animate` program from the ImageMagick suite is suitable, or one can simply show the file in a web page with the HTML command `<img src="tmpmovie.gif">`. AVI and MPEG files can be played by, for example, the `myplayer`, `vlc`, or `totem` programs.

### 4.3.8 Advanced Easyviz Topics

The information in the previous sections aims at being sufficient for the daily work with plotting curves. Sometimes, however, one wants to

fine-control the plot or how Easyviz behaves. First, we explain how to set the backend. Second, we tell how to speed up the `WILL BE REPLACED BY ptex2tex from scitools.std import *` statement. Third, we show how to operate with the plotting program directly and using plotting program-specific advanced features. Fourth, we explain how the user can grab `Figure` and `Axis` objects that Easyviz produces "behind the curtain".

*Controlling the Backend.* The Easyviz backend can either be set in a config file (see Config File below), by importing a special backend in the program, or by adding a command-line option

```
--SCITOOLS_easyviz_backend name
```

where `name` is the name of the backend: `gnuplot`, `vtk`, `matplotlib`, etc. Which backend you choose depends on what you have available on your computer system and what kind of plotting functionality you want.

An alternative method is to import a specific backend in a program. Instead of the `from scitools.std import *` statement one writes

```
from numpy import *
from scitools.easyviz.gnuplot_ import *  # work with Gnuplot
# or
from scitools.easyviz.vtk_ import *      # work with VTK
```

Note the trailing underscore in the module names for the various backends.

Easyviz is a subpackage of SciTools, and the the SciTools configuration file, called `scitools.cfg` has a section `[easyviz]` where the `backend` in Easyviz can be set:

```
[easyviz]
backend = vtk
```

A `.scitools.cfg` file can be placed in the current working folder, thereby affecting plots made in this folder, or it can be located in the user's home folder, which will affect all plotting sessions for the user in question. There is also a common SciTools config file `scitools.cfg` for the whole site (located in the directory where the `scitools` package is installed).

The following program prints a list of the names of the available backends on your computer system:

```
from scitools.std import *
backends = available_backends()
print 'Available backends:', backends
```

There will be quite some output explaining the missing backends and what must be installed to use these backends.

*Importing Just Easyviz.* The `from scitools.std import *` statement imports many modules and packages::

```
from numpy import *
from scitools.numpyutils import *  # some convenience functions
from numpy.lib.scimath import *
from scipy import *                # if scipy is installed
import sys, operator, math
from scitools.StringFunction import StringFunction
from glob import glob
```

The `scipy` import can take some time and lead to slow start-up of plot scripts. A more minimalistic import for curve plotting is

```
from scitools.easyviz import *
from numpy import *
```

Alternatively, one can edit the `scitools.cfg` configure file or add one's own `.scitools.cfg` file with redefinition of selected options, such as `load` in the `scipy` section. The user `.scitools.cfg` must be placed in the folder where the plotting script in action resides, or in the user's home folder. Instead of editing a configuration file, one can just add the command-line argument `--SCITOOLS_scipy_load no` to the curve plotting script (all sections/options in the configuration file can also be set by such command-line arguments).

*Working with the Plotting Program Directly.* Easyviz supports just the most common plotting commands, typically the commands you use "95 percent" of the time when exploring curves. Various plotting packages have lots of additional commands for different advanced features. When Easyviz does not have a command that supports a particular feature, one can grab the Python object that communicates with the underlying plotting program (known as "backend") and work with this object directly, using plotting program-specific command syntax. Let us illustrate this principle with an example where we add a text and an arrow in the plot, see Figure 4.9.

Easyviz does not support arrows at arbitrary places inside the plot, but Gnuplot does. If we use Gnuplot as backend, we may grab the `Gnuplot` object and issue Gnuplot commands to this object directly. Here is an example of the typical recipe, written after the core of the plot is made in the ordinary (plotting program-independent) way:

```
g = get_backend()
if backend == 'gnuplot':
    # g is a Gnuplot object, work with Gnuplot commands directly:
    g('set label "global maximum" at 0.1,0.5 font "Times,18"')
    g('set arrow from 0.5,0.48 to 0.98,0.37 linewidth 2')
    g.refresh()
    g.hardcopy('tmp2.eps')  # make new hardcopy
```

We refer to the Gnuplot manual for the features of this package and the syntax of the commands. The idea is that you can quickly generate plots with Easyviz using standard commands that are independent of the underlying plotting package. However, when you need advanced

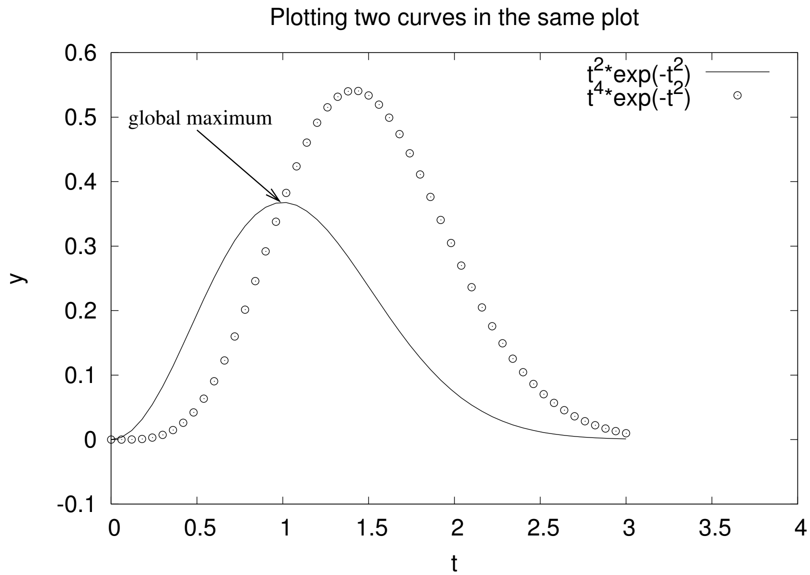



**Fig. 4.9** Illustration of a text and an arrow using Gnuplot-specific commands.

features, you must add plotting package-specific code as shown above. This principle makes Easyviz a light-weight interface, but without limiting the available functionality of various plotting programs.

*Working with Axis and Figure Objects.* Easyviz supports the concept of Axis objects, as in Matlab. The Axis object represents a set of axes, with curves drawn in the associated coordinate system. A figure is the complete physical plot. One may have several axes in one figure, each axis representing a subplot. One may also have several figures, represented by different windows on the screen or separate hardcopies.

Users with Matlab experience may prefer to set axis labels, ranges, and the title using an Axis object instead of providing the information in separate commands or as part of a `plot` command. The `gca` (get current axis) command returns an `Axis` object, whose `set` method can be used to set axis properties:

```
plot(t, y1, 'r-', t, y2, 'bo',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     hardcopy='tmp2.eps')

ax = gca()   # get current Axis object
ax.setp(xlabel='t', ylabel='y',
        axis=[0, 4, -0.1, 0.6],
        title='Plotting two curves in the same plot')
show()  # show the plot again after ax.setp actions
```

The `figure()` call makes a new figure, i.e., a new window with curve plots. Figures are numbered as 1, 2, and so on. The command `figure(3)` sets the current figure object to figure number 3.

Suppose we want to plot our y1 and y2 data in two separate windows. We need in this case to work with two Figure objects:

```
plot(t, y1, 'r-', xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6])

figure()  # new figure

plot(t, y2, 'bo', xlabel='t', ylabel='y')
```

We may now go back to the first figure (with the y1 data) and set a title and legends in this plot, show the plot, and make a PostScript version of the plot:

```
figure(1)  # go back to first figure
title('One curve')
legend('t^2*exp(-t^2)')
show()
hardcopy('tmp2_1.eps')
```

We can also adjust figure 2:

```
figure(2)  # go to second figure
title('Another curve')
hardcopy('tmp2_2.eps')
show()
```

The current Figure object is reached by gcf (get current figure), and the dump method dumps the internal parameters in the Figure object:

```
fig = gcf(); print fig.dump()
```

These parameters may be of interest for troubleshooting when Easyviz does not produce what you expect.

Let us then make a third figure with two plots, or more precisely, two axes: one with y1 data and one with y2 data. Easyviz has a command subplot(r,c,a) for creating r rows and c columns and set the current axis to axis number a. In the present case subplot(2,1,1) sets the current axis to the first set of axis in a "table" with two rows and one column. Here is the code for this third figure:

```
figure()  # new, third figure
# plot y1 and y2 as two axis in the same figure:
subplot(2, 1, 1)
plot(t, y1, xlabel='t', ylabel='y')
subplot(2, 1, 2)
plot(t, y2, xlabel='t', ylabel='y')
title('A figure with two plots')
show()
hardcopy('tmp2_3.eps')
```

If we need to place an axis at an arbitrary position in the figure, we must use the command

```
ax = axes(viewport=[left, bottom, width, height])
```

The four parameteres `left`, `bottom`, `width`, `height` are location values between 0 and 1 ((0,0) is the lower-left corner and (1,1) is the upper-right corner). However, this might be a bit different in the different backends (see the documentation for the backend in question).

### 4.3.9 Curves in Pure Text

Sometimes it can be desirable to show a graph in pure ASCII text, e.g., as part of a trial run of a program included in the program itself (cf. the introduction to Chapter 1.8), or a graph can be needed in a doc string. For such purposes we have slightly extended a module by Imri Goldberg (`aplotter.py`) and included it as a module in SciTools. Running `pydoc` on `scitools.aplotter` describes the capabilities of this type of primitive plotting. Here we just give an example of what it can do:

```
>>> from scitools.aplotter import plot
>>> from numpy import linspace, exp, cos, pi
>>> x = linspace(-2, 2, 81)
>>> y = exp(-0.5*x**2)*cos(pi*x)
>>> plot(x, y)
                                        |
                                     // |\\
                                    /   |  \
                                   /    |   \
                                  /     |   \
                                  /     |    \
                                 /      |     \
                                /       |      \
                               /        |      \
   -------\                    /        |       \
---+-------\\-----------------/---------+--------\-----------------
   -2        \                /         |        \                /
              \\              /         |         \             //
                \            /          |         \            /
                 \\         /           |          \         //
                   \       /            |           \       /
                    \    //             |            \-    //
                     ----               -0.63          ---/
                                        |

>>> # plot circles at data points only:
>>> plot(x, y, dot='o', plot_slope=False)
                                        |
                                       o+1
                                     oo |oo
                                    o   |  o
                                   o    |   o
                                        |
                                 o      |     o
                                o       |      o
                                        |
   oooooooo                    o        |       o
---+-------oo-----------------o---------+--------o-----------------
   -2        o                          |                         o
              oo              o         |         o             oo
```

```
                 o            o            |            o            o
                  oo         o             |             o          oo
                    o       o              |              o        o
                     o    oo               |               oo    oo
                      oooo                 -0.63              oooo
                                           |
>>> p = plot(x, y, output=str)   # store plot in a string:
>>> print p
```

(The last 13 characters of the output lines are here removed to make the lines fit the maximum textwidth of this book.)

## 4.4 Plotting Difficulties

The previous examples on plotting functions demonstrate how easy it is to make graphs. Nevertheless, the shown techniques might easily fail to plot some functions correctly unless we are careful. Next we address two types of difficult functions: piecewisely defined functions and rapidly varying functions.

### 4.4.1 Piecewisely Defined Functions

A piecewisely defined function has different function definitions in different intervals along the $x$ axis. The resulting function, made up of pieces, may have discontinuities in the function value or in derivatives. We have to be very careful when plotting such functions, as the next two examples will show. The problem is that the plotting mechanism draws straight lines between coordinates on the function's curve, and these straight lines may not yield a satisfactory visualization of the function. The first example has a discontinuity in the function itself at one point, while the other example has a discontinuity in the derivative at three points.

*Example: The Heaviside Function.* Let us plot the Heaviside function defined in (2.18) on page 108. The most natural way to proceed is first to define the function as

```
def H(x):
    return (0 if x < 0 else 1)
```

The standard plotting procedure where we define a coordinate array `x` and do a

```
y = H(x)
plot(x, y)
```

fails with this `H(x)` function. The test `x < 0` results in an array where each element is `True` or `False` depending on whether `x[i] < 0` or not.

A `ValueError` exception is raised when we use this resulting array in an `if` test:

```
>>> x = linspace(-10, 10, 5)
>>> x
array([-10.,  -5.,   0.,   5.,  10.])
>>> b = x < 0
>>> b
array([ True,  True, False, False, False], dtype=bool)
>>> bool(b)  # evaluate b in a boolean context
...
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

The suggestion of using the `any` or `all` methods do not help because this is not what we are interested in:

```
>>> b.any()  # True if any element in b is True
True
>>> b.all()  # True if all elements in b are True
False
```

We want to take actions element by element depending on whether `x[i] < 0` or not.

There are three ways to find a remedy to our problems with the `if x < 0` test: (i) we can write an explicit loop for computing the elements, (ii) we can use a tool for automatically vectorize `H(x)`, or (iii) we can manually vectorize the `H(x)` function. All three methods will be illustrated next.

*Loop.* The following function works well for arrays if we insert a simple loop over the array elements (such that `H(x)` operates on scalars only):

```
def H_loop(x):
    r = zeros(len(x))
    for i in xrange(len(x)):
        r[i] = H(x[i])
    return r

x = linspace(-5, 5, 6)
y = H_loop(x)
```

This `H_loop` version of `H` is sufficient for plotting the Heaviside function. The next paragraph explains other ways of making versions of `H(x)` that work for array arguments.

*Automatic Vectorization.* Numerical Python contains a method for automatically vectorizing a Python function that works with scalars (pure numbers) as arguments:

```
from numpy import vectorize
H_vec = vectorize(H)
```

The `H_vec(x)` function will now work with vector/array arguments `x`. Unfortunately, such automatically vectorized functions are often as slow as the explicit loop shown above.

*Manual Vectorization.* (Note: This topic is considered advanced and at another level than the rest of the book.) To allow array arguments in our Heaviside function *and* get the increased speed that one associates with vectorization, we have to rewrite the `H` function completely. The mathematics must now be expressed by functions from the Numerical Python library. In general, this type of rewrite is non-trivial and requires knowledge of and experience with the library. Fortunately, functions of the form

```
def f(x):
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x
```

can in general be vectorized quite simply as

```
def f_vectorized(x):
    x1 = <expression1>
    x2 = <expression2>
    return where(condition, x1, x2)
```

The `where` function returns an array of the same length as `condition`, and element no. `i` equals `x1[i]` if `condition[i]` is `True`, and `x2[i]` otherwise. With Python loops we can express this principle as

```
r = zeros(len(condition))   # array returned from where(...)
for i in xrange(condition):
    r[i] = x1[i] if condition[i] else x2[i]
```

The `x1` and `x2` variables can be pure numbers or arrays of the same length as `x`.

In our case we can use the `where` function as follows:

```
def Hv(x):
    return where(x < 0, 0.0, 1.0)
```

*Plotting the Heaviside Function.* Since the Heaviside function consists of two flat lines, one may think that we do not need many points along the $x$ axis to describe the curve. Let us try only five points:

```
x = linspace(-10, 10, 5)
plot(x, Hv(x), axis=[x[0], x[-1], -0.1, 1.1])
```

However, so few $x$ points are not able to describe the jump from 0 to 1 at $x = 0$, as shown by the solid line in Figure 4.10a. Using more points, say 50 between $-10$ and 10,

```
x2 = linspace(-10, 10, 50)
plot(x, Hv(x), 'r', x2, Hv(x2), 'b',
     legend=('5 points', '50 points'),
     axis=[x[0], x[-1], -0.1, 1.1])
```

makes the curve look better, as you can see from the dotted line in Figure 4.10a. However, the step is still not vertical. This time the point $x = 0$ was not among the coordinates so the step goes from $x = -0.2$ to $x = 0.2$. More points will improve the situation. Nevertheless, the best is to draw two flat lines directly: from $(-10, 0)$ to $(0, 0)$, then to $(0, 1)$ and then to $(10, 1)$:

```
plot([-10, 0, 0, 10], [0, 0, 1, 1],
     axis=[x[0], x[-1], -0.1, 1.1])
```

The result is shown in Figure 4.10b.

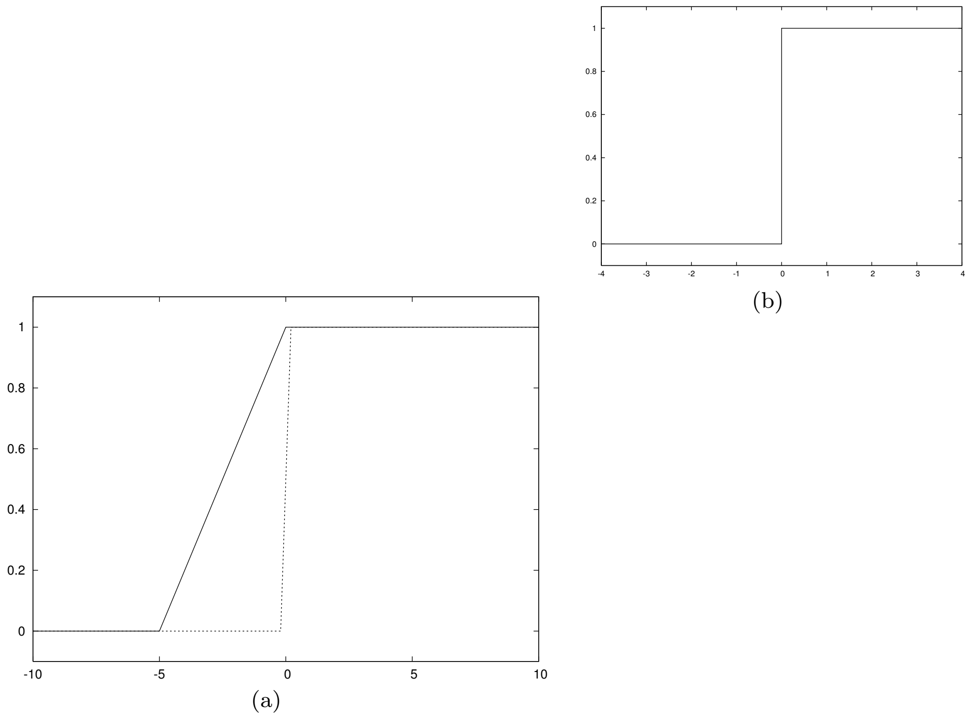



**Fig. 4.10** Plot of the Heaviside function: (a) using equally spaced $x$ points (5 and 50); (b) using a "double point" at $x = 0$.

Some will argue that the plot of $H(x)$ should not contain the vertical line from $(0, 0)$ to $(0, 1)$, but only two horizontal lines. To make such a plot, we must draw two distinct curves, one for each horizontal line:

```
plot([-10,0], [0,0], 'r-', [0,10], [1,1], 'r-',
     axis=[x[0], x[-1], -0.1, 1.1])
```

Observe that we must specify the same line style for both lines (curves), otherwise they would by default get different color on the screen and different line type in a hardcopy. We remark, however, that discontinuous functions like $H(x)$ are often visualized with vertical lines at the jumps, as we do in Figure 4.10b.

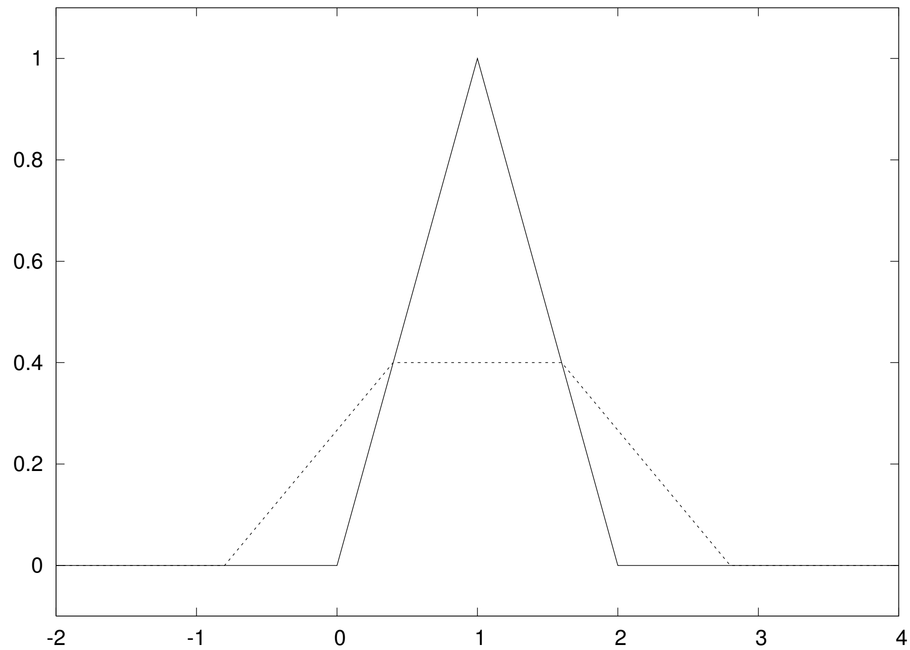



**Fig. 4.11** Plot of a "hat" function. The solid line shows the exact function, while the dashed line arises from using inappropriate points along the $x$ axis.

*Example: A "Hat" Function.* Let us plot the "hat" function $N(x)$, defined by (2.5) on page 89. The corresponding Python implementation `N(x)` shown right after (2.5) does not work with array arguments `x` because the boolean expressions, like `x < 0`, are arrays and they cannot yield a single `True` or `False` value for the `if` tests. The simplest solution is to use `vectorize`, as explained for the Heaviside function above[8]:

```
N_vec = vectorize(N)
```

A manual rewrite, yielding a faster vectorized function, is more demanding than for the Heaviside function because we now have multiple branches in the `if` test. One attempt may be[9]

```
def Nv(x):
    r = where(x < 0, 0.0, x)
    r = where(0 <= x < 1, x, r)
    r = where(1 <= x < 2, 2-x, r)
    r = where(x >= 2, 0.0, r)
    return r
```

However, the condition like `0 <= x < 1`, which is equivalent to `0 <= x and x < 1`, does not work because the `and` operator does not work with array arguments. All operators in Python (`+`, `-`, `and`, `or`, etc.)

[8] It is important that `N(x)` return `float` and not `int` values, otherwise the vectorized version will produce `int` values and hence be incorrect.

[9] This is again advanced material.

are available as pure functions in a module `operator` (`operator.add`, `operator.sub`, `operator.and_`, `operator.or_`[10], etc.). A working `Nv` function must apply `operator.and_` instead:

```
def Nv(x):
    r = where(x < 0, 0.0, x)
    import operator
    condition = operator.and_(0 <= x, x < 1)
    r = where(condition, x, r)
    condition = operator.and_(1 <= x, x < 2)
    r = where(condition, 2-x, r)
    r = where(x >= 2, 0.0, r)
    return r
```

A second, alternative rewrite is to use boolean expressions in indices:

```
def Nv(x):
    r = x.copy()  # avoid modifying x in-place
    r[x < 0.0] = 0.0
    condition = operator.and_(0 <= x, x < 1)
    r[condition] = x[condition]
    condition = operator.and_(1 <= x, x < 2)
    r[condition] = 2-x[condition]
    r[x >= 2] = 0.0
    return r
```

Now to the computation of coordinate arrays for the plotting. We may use an explicit loop over all array elements, or the `N_vec` function, or the `Nv` function. An approach without thinking about vectorization too much could be

```
x = linspace(-2, 4, 6)
plot(x, N_vec(x), 'r', axis=[x[0], x[-1], -0.1, 1.1])
```

This results in the dashed line in Figure 4.11. What is the problem? The problem lies in the computation of the `x` vector, which does not contain the points $x = 1$ and $x = 2$ where the function makes significant changes. The result is that the "hat" is "flattened". Making an `x` vector with all critical points in the function definitions, $x = 0, 1, 2$, provides the necessary remedy, either

```
x = linspace(-2, 4, 7)
```

or the simple

```
x = [-2, 0, 1, 2, 4]
```

Any of these `x` alternatives and a `plot(x, N_vec(x))` will result in the solid line in Figure 4.11, which is the correct visualization of the $N(x)$ function.

[10] Recall that `and` and `or` are reserved keywords, see page 10, so a module or program cannot have variables or functions with these names. To circumvent this problem, the convention is to add a trailing underscore to the name.

### 4.4.2 Rapidly Varying Functions

Let us now visualize the function $f(x) = \sin(1/x)$, using 10 and 1000 points:

```
def f(x):
    return sin(1.0/x)

x1 = linspace(-1, 1, 10)
x2 = linspace(-1, 1, 1000)
plot(x1, f(x1), label='%d points' % len(x))
plot(x2, f(x2), label='%d points' % len(x))
```

The two plots are shown in Figure 4.12. Using only 10 points gives a completely wrong picture of this function, because the function oscillates faster and faster as we approach the origin. With 1000 points we get an impression of these oscillations, but the accuracy of the plot in the vicinity of the origin is still poor. A plot with 100000 points has better accuracy, in principle, but the extermely fast oscillations near the origin just drowns in black ink (you can try it out yourself).

Another problem with the $f(x) = \sin(1/x)$ function is that it is easy to define an `x` vector that contains $x = 0$, such that we get division by zero. Mathematically, the $f(x)$ function has a singularity at $x = 0$: it is difficult to define $f(0)$, so one should exclude this point from the function definition and work with a domain $x \in [-1, -\epsilon] \cup [\epsilon, 1]$, with $\epsilon$ chosen small.

The lesson learned from these examples is clear: You must investigate the function to be visualized and make sure that you use an appropriate set of $x$ coordinates along the curve. A relevant first step is to double the number of $x$ coordinates and check if this changes the plot. If not, you probably have an adequate set of $x$ coordinates.

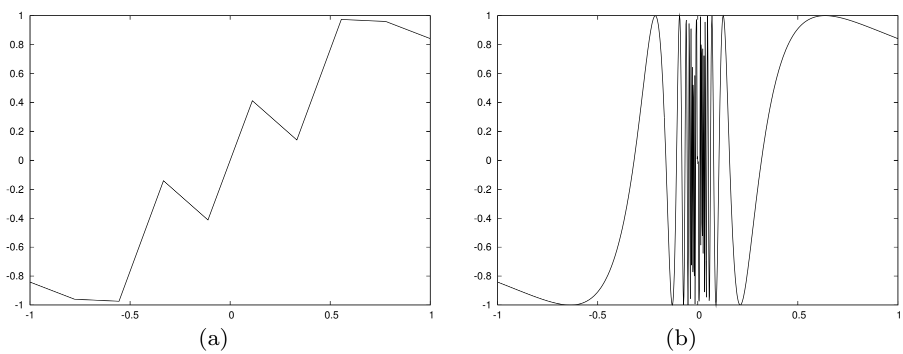



**Fig. 4.12** Plot of the function $\sin(1/x)$ with (a) 10 points and (b) 1000 points.

### 4.4.3 Vectorizing StringFunction Objects

The `StringFunction` object described in Chapter 3.1.4 does unfortunately not work with array arguments unless we explicitly tell the object to do so. The recipe is very simple. Say `f` is some `StringFunction` object. To allow array arguments we must first call `f.vectorize(globals())` once:

```
f = StringFunction(formula)
x = linspace(0, 1, 30)

# f(x) will in general not work

from numpy import *
f.vectorize(globals())
# now f works with array arguments:
values = f(x)
```

It is important that you import everything from `numpy` or `scitools.std` *before* you call `f.vectorize`. We suggest to take the `f.vectorize` call as a magic recipe[11].

Even after calling `f.vectorize(globals())` a `StringFunction` object may face problems with vectorization. One example is a piecewise constant function as specified by a string expression `'1 if x > 2 else 0'`. One remedy is to use the vectorized version of an `if` test: `'where(x > 2, 1, 0)'`. For an average user of the program this construct is not at all obvious so a more user-friendly solution is to apply `vectorize` from `numpy`:

```
f = vectorize(f)  # vectorize a StringFunction f
```

The line above is therefore the most general (but also the slowest) way of vectorizing a `StringFunction` object. After that call, `f` is no more a `StringFunction` object, but `f` behaves as a (vectorized) function. The `vectorize` tool from `numpy` can be used to allow any Python function taking scalar arguments to also accept array arguments.

To get better speed, one can use `vectorize(f)` only in the case the formula in `f` contains an inline `if` test (e.g., recognized by the string `' else '` inside the formula). Otherwise, we use `f.vectorize`. The formula in `f` is obtained by `str(f)` so we can test

[11] Some readers still want to know what the problem is. Inside the `StringFunction` module we need to have access to mathematical functions for expressions like `sin(x)*exp(x)` to be evaluated. These mathematical functions are by default taken from the `math` module and hence they do not work with array arguments. If the user, in the main program, has imported mathematical functions that work with array arguments, these functions are registered in a dictionary returned from `globals()`. By the `f.vectorize` call we supply the `StringFunction` module with the user's global namespace so that the evaluation of the string expression can make use of mathematical functions for arrays.

```
if ' else ' in str(f):
    f = vectorize(f)
else:
    f.vectorize(globals())
```

## 4.5 More on Numerical Python Arrays

This section lists some more advanced but useful operations with Numerical Python arrays.

### 4.5.1 Copying Arrays

Let `x` be an array. The statement `a = x` makes `a` refer to the same array as `x`. Changing `a` will then also affect `x`:

```
>>> x = array([1, 2, 3.5])
>>> a = x
>>> a[-1] = 3  # this changes x[-1] too!
>>> x
array([ 1.,  2.,  3.])
```

Changing `a` without changing `x` requires `a` to be a copy of `x`:

```
>>> a = x.copy()
>>> a[-1] = 9
>>> a
array([ 1.,  2.,  9.])
>>> x
array([ 1.,  2.,  3.])
```

### 4.5.2 In-Place Arithmetics

Let `a` and `b` be two arrays of the same shape. The expression `a += b` means `a = a + b`, but this is not the complete story. In the statement `a = a + b`, the sum `a + b` is first computed, yielding a new array, and then the name `a` is bound to this new array. The old array `a` is lost unless there are other names assigned to this array. In the statement `a += b`, elements of `b` are added directly into the elements of `a` (in memory). There is no hidden intermediate array as in `a = a + b`. This implies that `a += b` is more efficient than `a = a + b` since Python avoids making an extra array. We say that the operators `+=`, `*=`, and so on, perform *in-place* arithmetics in arrays.

Consider the compound array expression

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

The computation actually goes as follows with seven hidden arrays for storing intermediate results:

1. `r1 = x**4`
2. `r2 = 3*r1`
3. `r3 = 2*x`
4. `r4 = r2 + r3`
5. `r5 = r4 + 4`
6. `r6 = x + 1`
7. `r7 = r5/r6`
8. `a = r7`

With in-place arithmetics we can get away with creating three new arrays, at a cost of a significantly less readable code:

```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

The three extra arrays in this series of statement arise from copying `x`, and computing the right-hand sides `2*x` and `x+1`.

Quite often in computational science and engineering, a huge number of arithmetics is performed on very large arrays, and then saving memory and array allocation time by doing in-place arithmetics is important.

The mix of assignment and in-place arithmetics makes it easy to make unintended changes of more than one array. For example, this code changes `x`:

```
a = x
a += y
```

since `a` refers to the same array as `x` and the change of `a` is done in-place.

### 4.5.3 Allocating Arrays

We have already seen that the `zeros` function is handy for making a new array `a` of a given size. Very often the size and the type of array elements have to match another existing array `x`. We can then either copy the original array, e.g.,

```
a = x.copy()
```

and fill elements in `a` with the right new values, or we can say

```
a = zeros(x.shape, x.dtype)
```

The attribute `x.dtype` holds the array element type (`dtype` for data type), and as mentioned before, `x.shape` is a tuple with the array dimensions.

Sometimes we may want to ensure that an object is an array, and if not, turn it into an array. The `asarray` function is useful in such cases:

```
a = asarray(a)
```

Nothing is copied if `a` already is an array, but if `a` is a list or tuple, a new array with a copy of the data is created.

### 4.5.4 Generalized Indexing

Chapter 4.2.2 shows how slices can be used to extract and manipulate subarrays. The slice `f:t:i` corresponds to the index set `f, f+i, f+2*i,` ... up to, but not including, `t`. Such an index set can be given explicitly too: `a[range(f,t,i)]`. That is, the integer list from `range` can be used as a set of indices. In fact, any integer list or integer array can be used as index:

```
>>> a = linspace(1, 8, 8)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])
>>> a[range(2,8,3)] = -2   # same as a[2:8:3] = -2
>>> a
array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
```

We can also use boolean arrays to generate an index set. The indices in the set will correspond to the indices for which the boolean array has `True` values. This functionality allows expressions like `a[x<m]`. Here are two examples, continuing the previous interactive session:

```
>>> a[a < 0]            # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
>>> # replace elements where a is 10 by the first
>>> # elements from another array/list:
>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]
>>> a
array([  1.,  10.,  20.,   4.,   5.,  30.,  40.,  50.])
```

Generalized indexing using integer arrays or lists is important for vectorized initialization of array elements.

### 4.5.5 Testing for the Array Type

Inside an interactive Python shell you can easily check an object's type using the `type` function (see Chapter 1.5.2). In case of a Numerical Python array, the type name is `ndarray`:

```
>>> a = linspace(-1, 1, 3)
>>> a
array([-1.,  0.,  1.])
>>> type(a)
<type 'numpy.ndarray'>
```

Sometimes you need to test if a variable is an `ndarray` or a `float` or `int`. The `isinstance` function was made for this purpose:

```
>>> isinstance(a, ndarray)
True
>>> type(a) == ndarray
True
>>> isinstance(a, (float,int))  # float or int?
False
```

A typical use of `isinstance` is shown next.

*Example: Vectorizing a Constant Function.* Suppose we have a constant function,

```
def f(x):
    return 2
```

This function accepts an array argument `x`, but will return a `float` while a vectorized version of the function should return an array of the same shape as `x` where each element has the value 2. The vectorized version can be realized as

```
def fv(x):
    return zeros(x.shape, x.dtype) + 2
```

The optimal vectorized function would be one that works for both a scalar and an array argument. We must then test on the argument type:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    else:  # assmue array
        return zeros(x.shape, x.dtype) + 2
```

A more foolproof solution is to also test for an array and raise an exception if `x` is neither a scalar nor an array:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    elif isinstance(x, ndarray):
        return zeros(x.shape, x.dtype) + 2
```

```
    else:
        raise TypeError\
        ('x must be int, float or ndarray, not %s' % type(x))
```

### 4.5.6 Equally Spaced Numbers

We have used the `linspace` function heavily so far in this chapter, but there are some related, useful functions that also produce a sequence of uniformly spaced numbers. In `numpy` we have the `arange` function, where `arange(start, stop, inc)` creates an array with the numbers `start, start+inc, start+2*inc, ..., stop-inc`. Note that the upper limit `stop` is not included in the set of numbers (i.e., `arange` behaves as `range` and `xrange`):

```
>>> arange(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5])
```

Because of round-off errors the upper limit can be included in the array. You can try out

```
for i in range(1,500):
    a = arange(0, 1, 1.0/i)
    print i, a[-1]
```

Now and then, the last element `a[-1]` equals 1, which is wrong behavior! We therefore recommend to stay away from `arange`. A substitute for `arange` is the function `seq` from SciTools: `seq(start, stop, inc)` generates real numbers starting with `start`, ending with `stop`, and with increments of `inc`.

```
>>> from scitools.std import *
>>> seq(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

For integers, a similar function, called `iseq`, is available. With `iseq(start, stop, inc)` we get the integers `start`, `start+inc`, `start+2*inc`, and so on up to `stop`. Unlike `range(start, stop, inc)` and `xrange(start, stop, inc)`, the upper limit `stop` is part of the sequence of numbers. This feature makes `iseq` more appropriate than `range` and `xrange` in many implementations of mathematical algorithms where there is an index whose limits are specified in the algorithm, because with `iseq` we get a one-to-one correspondence between the algorithm and the Python code. Here is an example: a vector $x$ of length $n$, compute

$$a_i = f(x_{i+1}) - f(x_{i-1}) \text{ for } i = 1, \ldots, n-2\,.$$

A Python implementation with `iseq` reads

```
for i in iseq(1, n-2):
    a[i] = f(x[i+1]) - f(x[i-1])
```

while with `range` we must write

```
for i in range(1, n-1):
    a[i] = f(x[i+1]) - f(x[i-1])
```

Direct correspondence between the mathematics and the code is very important and makes it much easier to find bugs by comparing the code and the mathematical description, line by line.

### 4.5.7 Compact Syntax for Array Generation

There is a special compact syntax `r_[f:t:s]` for the `linspace` and `arange` functions:

```
>>> a = r_[-5:5:11j]  # same as linspace(-5, 5, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

Here, `11j` means 11 coordinates (between -5 and 5, including the upper limit 5). That is, the number of elements in the array is given with the imaginary number syntax.

The `arange` equivalent reads

```
>>> a = r_[-5:5:1.0]
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

With `1` as step instead of `1.0` (`r_[-5:5:1]`) the elements in `a` become integers.

### 4.5.8 Shape Manipulation

The `shape` attribute in array objects holds the shape, i.e., the size of each dimension. A function `size` returns the total number of elements in an array. Here are a few equivalent ways of changing the shape of an array:

```
>>> a = linspace(-1, 1, 6)
>>> a.shape
(6,)
>>> a.size
6
>>> a.shape = (2, 3)
>>> a.shape
(2, 3)
>>> a.size                  # total no of elements
6
>>> a.shape = (a.size,)     # reset shape
>>> a = a.reshape(3, 2)     # alternative
>>> len(a)                  # no of rows
3
```

Note that `len(a)` always returns the length of the first dimension of an array.

## 4.6 Higher-Dimensional Arrays

### 4.6.1 Matrices and Arrays

Vectors appeared when mathematicians needed to calculate with a list of numbers. When they needed a table (or a list of lists in Python terminology), they invented the concept of *matrix* (singular) and *matrices* (plural). A table of numbers has the numbers ordered into rows and columns. One example is

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

This table with three rows and four columns is called a $3 \times 4$ matrix[12]. If the symbol $A$ is associated with this matrix, $A_{i,j}$ denotes the number in row number $i$ and column number $j$. Counting rows and columns from 0, we have, for instance, $A_{0,0} = 0$ and $A_{2,3} = -2$. We can write a general $m \times n$ matrix $A$ as

$$\begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Matrices can be added and subtracted. They can also be multiplied by a scalar (a number), and there is a concept of "length". The formulas are quite similar to those presented for vectors, but the exact form is not important here.

We can generalize the concept of table and matrix to *array*, which holds quantities with in general $d$ indices. Equivalently we say that the array has rank $d$. For $d = 3$, an array $A$ has elements with three indices: $A_{p,q,r}$. If $p$ goes from 0 to $n_p - 1$, $q$ from 0 to $n_q - 1$, and $r$ from 0 to $n_r - 1$, the $A$ array has $n_p \times n_q \times n_r$ elements in total. We may speak about the *shape* of the array, which is a $d$-vector holding the number of elements in each "array direction", i.e., the number of elements for each index. For the mentioned $A$ array, the shape is $(n_p, n_q, n_r)$.

The special case of $d = 1$ is a vector, and $d = 2$ corresponds to a matrix. When we program we may skip thinking about vectors and matrices (if you are not so familiar with these concepts from a mathematical point of view) and instead just work with arrays. The number

[12] Mathematicians don't like this sentence, but it suffices for our purposes.

of indices corresponds to what is convenient in the programming problem we try to solve.

### 4.6.2 Two-Dimensional Numerical Python Arrays

Recall the nested list from Chapter 2.1.7, where `[C, F]` pairs are elements in a list `table`. The construction of `table` goes as follows:

```
>>> Cdegrees = [-30 + i*10 for i in range(3)]
>>> Fdegrees = [9./5*C + 32 for C in Cdegrees]
>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
>>> print table
[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
```

Note that the `table` list is a nested list. This nested list can be turned into an array,

```
>>> table2 = array(table)
>>> print table2
[[-30. -22.]
 [-20.  -4.]
 [-10.  14.]]
>>> type(table2)
<type 'numpy.ndarray'>
```

We say that `table2` is a *two-dimensional* array, or an array of rank 2.

The `table` list and the `table2` array are stored very differently in memory. The `table` variable refers to a list object containing three elements. Each of these elements is a reference to a separate list object with two elements, where each element refers to a separate `float` object. The `table2` variable is a reference to a single array object that again refers to a consecutive sequence of bytes in memory where the six floating-point numbers are stored. The data associated with `table2` are found in one “chunk” in the computer’s memory, while the data associated with `table` are scattered around in memory. On today’s machines, it is much more expensive to find data in memory than to compute with the data. Arrays make the data fetching more efficient, and this is major reason for using arrays. However, this efficiency gain is only present for very large arrays, not for a $3 \times 2$ array.

Indexing a nested list is done in two steps, first the outer list is indexed, giving access to an element that is another list, and then this latter list is indexed:

```
>>> table[1][0]     # table[1] is [-20,4], whose index 0 holds -20
-20
```

This syntax works for two-dimensional arrays too:

```
>>> table2[1][0]
-20.0
```

but there is another syntax which is more common for arrays:

```
>>> table2[1,0]
-20.0
```

A two-dimensional array reflects a table and has a certain number of “rows” and “columns”. We refer to “rows” as the *first dimension* of the array and “columns” as the *second dimension*. These two dimensions are available as `table2.shape`:

```
>>> table2.shape
(3, 2)
```

Here, 3 is the number of “rows” and 2 is the number of “columns”.

A loop over all the elements in a two-dimensional array is usually expressed as two *nested* `for` loops, one for each index:

```
>>> for i in range(table2.shape[0]):
...     for j in range(table2.shape[1]):
...         print 'table2[%d,%d] = %g' % (i, j, table2[i,j])
...
table2[0,0] = -30
table2[0,1] = -22
table2[1,0] = -20
table2[1,1] = -4
table2[2,0] = -10
table2[2,1] = 14
```

An alternative (but less efficient) way of visiting each element in an array with any number of dimensions makes use of a single `for` loop:

```
>>> for index_tuple, value in ndenumerate(table2):
...     print 'index %s has value %g' % \
...           (index_tuple, table2[index_tuple])
...
index (0,0) has value -30
index (0,1) has value -22
index (1,0) has value -20
index (1,1) has value -4
index (2,0) has value -10
index (2,1) has value 14
```

In the same way as we can extract sublists of lists, we can extract subarrays of arrays using slices.

```
table2[0:table2.shape[0], 1]  # 2nd column (index 1)
array([-22.,  -4.,  14.])

>>> table2[0:, 1]             # same
array([-22.,  -4.,  14.])

>>> table2[:, 1]              # same
array([-22.,  -4.,  14.])
```

To illustrate array slicing further, we create a bigger array:

```
>>> t = linspace(1, 30, 30).reshape(5, 6)
>>> t
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
```

```
       [ 19.,  20.,  21.,  22.,  23.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])

>>> t[1:-1:2, 2:]
array([[  9.,  10.,  11.,  12.],
       [ 21.,  22.,  23.,  24.]])
```

To understand the slice, look at the original `t` array and pick out the two rows corresponding to the first slice `1:-1:2`,

```
[  7.,   8.,   9.,  10.,  11.,  12.]
[ 19.,  20.,  21.,  22.,  23.,  24.]
```

Among the rows, pick the columns corresponding to the second slice `2:`,

```
[  9.,  10.,  11.,  12.]
[ 21.,  22.,  23.,  24.]
```

Another example is

```
>>> t[:-2, :-1:2]
array([[  1.,   3.,   5.],
       [  7.,   9.,  11.],
       [ 13.,  15.,  17.]])
```

### 4.6.3 Array Computing

The operations on vectors in Chapter 4.1.3 can quite straightforwardly be extended to arrays of any dimension. Consider the definition of applying a function $f(v)$ to a vector $v$: we apply the function to each element $v_i$ in $v$. For a two-dimensional array $A$ with elements $A_{i,j}$, $i = 0, \ldots, m$, $j = 0, \ldots, n$, the same definition yields

$$f(A) = (f(A_{0,0}), \ldots, f(A_{m-1,0}), f(A_{1,0}), \ldots, f(A_{m-1,n-1}))\,.$$

For an array $B$ with any rank, $f(B)$ means applying $f$ to each array entry.

The asterix operation from Chapter 4.1.3 is also naturally extended to arrays: $A * B$ means multiplying an element in $A$ by the corresponding element in $B$, i.e., element $(i,j)$ in $A * B$ is $A_{i,j}B_{i,j}$. This definition naturally extends to arrays of any rank, provided the two arrays have the same shape.

Adding a scalar to an array implies adding the scalar to each element in the array. Compound expressions involving arrays, e.g., $\exp(-A * *2) * A + 1$, work as for vectors. One can in fact just imagine that all the array elements are stored after each other in a long vector[13], and the array operations can then easily be defined in terms of the vector operations from Chapter 4.1.3.

[13] This is the way the array elements are stored in the computer's memory.

*Remark.* Readers with knowlege of matrix computations may ask how an expression like $A^2$ interfere with $A{*}{*}2$. In matrix computing, $A^2$ is a matrix-matrix product, which is very different from squaring each element in $A$ as $A{*}{*}2$ or $A*A$ implies. Fortunately, the matrix computing operations look different from the array computing operations in mathematical typesetting. In a program, however, `A*A` and `A**2` are identical computations, but the first one could lead to a confusion with a matrix-matrix product $AA$. With Numerical Python the matrix-matrix product is obtained by `dot(A, A)`. The matrix-vector product $Ax$, where $x$ is a vector, is computed by `dot(A, x)`.

### 4.6.4 Two-Dimensional Arrays and Functions of Two Variables

Given a function of two variables, say

```
def f(x, y):
    return sin(sqrt(x**2 + y**2))
```

we can plot this function by writing

```
x = y = linspace(-5, 5, 21)  # coordinates in x and y direction
xv, yv = ndgrid(x, y)
z = f(xv, yv)
mesh(xv, yv, z)
```

There are two new things here: (i) the call to `ndgrid`, which is necessary to transform one-dimensional coordinate arrays in the $x$ and $y$ direction into arrays valid for evaluating `f` over a two-dimensional grid; and (ii) the plot function whose name now is `mesh`, which is one out of many plot functions for two-dimensional functions. Another plot type you can try out is

```
surf(xv, yv, z)
```

More material on visualizing $f(x, y)$ functions is found in the section "Visualizing Scalar Fields" in the Easyviz tutorial. This tutorial can be reached through the command `pydoc scitools.easyviz` in a terminal window.

### 4.6.5 Matrix Objects

This section only makes sense if you are familiar with basic linear algebra and the matrix concept. The arrays created so far have been of type `ndarray`. NumPy also has a matrix type called `matrix` or `mat` for one- and two-dimensional arrays. One-dimensional arrays are then extended with one extra dimension such that they become matrices, i.e., either a row vector or a column vector:

```
>>> x1 = array([1, 2, 3], float)
>>> x2 = matrix(x)                      # or mat(x)
>>> x2                                  # row vector
matrix([[ 1.,  2.,  3.]])
>>> x3 = mat(x).transpose()             # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.core.defmatrix.matrix'>
>>> isinstance(x3, matrix)
True
```

A special feature of `matrix` objects is that the multiplication operator represents the matrix-matrix, vector-matrix, or matrix-vector product as we know from linear algebra:

```
>>> A = eye(3)                          # identity matrix
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> A = mat(A)
>>> A
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> y2 = x2*A                           # vector-matrix product
>>> y2
matrix([[ 1.,  2.,  3.]])
>>> y3 = A*x3                           # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

One should note here that the multiplication operator between standard `ndarray` objects is quite different, as the next interactive session demonstrates.

```
>>> A*x1                                # no matrix-array product!
Traceback (most recent call last):
...
ValueError: matrices are not aligned

>>> # try array*array product:
>>> A = (zeros(9) + 1).reshape(3,3)
>>> A
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A*x1                            # [A[0,:]*x1, A[1,:]*x1, A[2,:]*x1]
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
>>> B = A + 1
>>> A*B                             # element-wise product
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
>>> A = mat(A);  B = mat(B)
```

```
>>> A*B                        # matrix-matrix product
matrix([[ 6.,  6.,  6.],
        [ 6.,  6.,  6.],
        [ 6.,  6.,  6.]])
```

Readers who are familiar with Matlab, or intend to use Python and Matlab together, should seriously think about programming with `matrix` objects instead of `ndarray` objects, because the `matrix` type behaves quite similar to matrices and vectors in Matlab. Nevertheless, `matrix` cannot be used for arrays of larger dimension than two.

## 4.7 Summary

### 4.7.1 Chapter Topics

This chapter has introduced computing with arrays and plotting curve data stored in arrays. The Numerical Python package contains lots of functions for array computing, including the ones listed in Table 4.1. The syntax for plotting curves makes use of Easyviz commands, which are very similar to those of Matlab, but a wide range of plotting packages can be used as "engines" for producing the graphics. Easyviz is a subpackage of the SciTools package, which is the key collection of software accompanying the present book.

*Array Computing.* When we apply a Python function `f(x)` to a Numerical Python array `x`, the result is the same as if we apply `f` to each element in `x` separately. However, when `f` contains `if` statements, these are in general invalid if an array `x` enters the boolean expression. We then have to rewrite the function, often by applying the `where` function from Numerical Python.

*Plotting Curves.* A typical Easyviz command for plotting some curves with control of curve legends, axis, plot title, and also making a file with the plot, can be illustrated by

```
from scitools.std import *   # get all we need for plotting

plot(t1, y1, 'r',  # curve 1, red line
     t2, y2, 'b',  # curve 2, blue line
     t3, y3, 'o',  # curve 3, circles at data points
     axis=[t1[0], t1[-1], -1.1, 1.1],
     legend=('model 1', 'model 2', 'measurements'),
     xlabel='time', ylabel='force',
     hardcopy='forceplot_%04d.png' % counter)
```

*Making Movies.* Each frame in a movie must be a hardcopy of a plot, i.e., a plotfile. These plotfiles should have names containing a counter padded with leading zeros. One example may be the name

**Table 4.1** Summary of important functionality for Numerical Python arrays.

| | |
|---|---|
| `array(ld)` | copy list data `ld` to a `numpy` array |
| `asarray(d)` | make array of data `d` (copy if list, no copy if already array) |
| `zeros(n)` | make a `float` vector/array of length `n`, with zeros |
| `zeros(n, int)` | make an `int` vector/array of length `n` with zeros |
| `zeros((m,n))` | make a two-dimensional `float` array with shape (`m,n`) |
| `zeros(x.shape, x.dtype)` | make array of same shape as `x` and same element data type |
| `linspace(a,b,m)` | uniform sequence of `m` numbers between `a` and `b` (`b` is included in the sequence) |
| `seq(a,b,h)` | uniform sequence of numbers from `a` to `b` with step `h` (SciTools specific, largest element is `>= b`) |
| `iseq(a,b,h)` | uniform sequence of integers from `a` to `b` with step `h` (SciTools specific, largest element is `>= b`) |
| `a.shape` | tuple containing `a`'s shape |
| `a.size` | total no of elements in `a` |
| `len(a)` | length of a one-dim. array `a` (same as `a.shape[0]`) |
| `a.reshape(3,2)` | return `a` reshaped as $2 \times 3$ array |
| `a[i]` | vector indexing |
| `a[i,j]` | two-dim. array indexing |
| `a[1:k]` | slice: reference data with indices `1`,...,`k-1` |
| `a[1:8:3]` | slice: reference data with indices `1`, `4`,...,`7` |
| `b = a.copy()` | copy an array |
| `sin(a), exp(a), ...` | `numpy` functions applicable to arrays |
| `c = concatenate(a, b)` | `c` contains `a` with `b` appended |
| `c = where(cond, a1, a2)` | `c[i] = a1[i]` if `cond[i]`, else `c[i] = a2[i]` |
| `isinstance(a, ndarray)` | is `True` if `a` is an array |

specified as the `hardcopy` argument in the `plot` command above: `forceplot_0001.eps`, `forceplot_0002.eps`, etc., if `counter` runs from 1. Having the plotfiles with names on this form, we can make a movie file `movie.gif` with two frames per second by

```
movie('forceplot_*.png', encoder='convert',
      output_file='movie.gif', fps=2)
```

The resulting movie, in the animated GIF format, can be shown in a web page or displayed by the `animate` program.

Other movie formats can be produced by using other encoders, e.g., `ppmtompeg` and `ffmpeg` for the MPEG format, or `mencoder` for the AVI format. There are lots of options to the `movie` function, which you can see by writing `pydoc scitools.easyviz.movie` (see page 98 for how to run such a command).

### 4.7.2 Summarizing Example: Animating a Function

*Problem.* In this chapter's summarizing example we shall visualize how the temperature varies downward in the earth as the surface temperature oscillates between high day and low night values. One question may be: What is the temperature change 10 m down in the ground if

the surface temperature varies between 2 C in the night and 15 C in the day?

Let the $z$ axis point downwards, towards the center of the earth, and let $z = 0$ correspond to the earth's surface. The temperature at some depth $z$ in the ground at time $t$ is denoted by $T(z,t)$. If the surface temperature has a periodic variation around some mean value $T_0$, according to

$$T(0,t) = T_0 + A\cos(\omega t),$$

one can find, from a mathematical model for heat conduction, that the temperature at an arbitrary depth is

$$T(z,t) = T_0 + Ae^{-az}\cos(\omega t - az), \quad a = \sqrt{\frac{\omega}{2k}}\,. \qquad (4.13)$$

The parameter $k$ reflects the ground's ability to conduct heat ($k$ is called the *heat conduction coefficient*).

The task is to make an animation of how the temperature profile in the ground, i.e., $T$ as a function of $z$, varies in time. Let $\omega$ correspond to a time period of 24 hours. The mean temperature $T_0$ at the surface can be taken as 10 C, and the maximum variation $A$ can be set to 10 C. The heat conduction coefficient $k$ equals 1 $\text{mm}^2/\text{s}$ (which is $10^{-6}$ $\text{m}^2/\text{s}$ in proper SI units).

*Solution.* To animate $T(z,t)$ in time, we need to make a loop over points in time, and in each pass in the loop we must make a hardcopy of the plot of $T$ as a function of $z$. The files with the hardcopies can then be combined to a movie. The algorithm becomes

> for $t_i = i\Delta t$, $i = 0, 1, 2\ldots, n$:
>     plot the curve $y(z) = T(z,t_i)$
>     make hardcopy (store the plot in a file)
> combine all the plot files into a movie

It can be wise to make a general `animate` function where we just feed in some $f(x,t)$ function and make all the plot files. If `animate` has arguments for setting the labels on the axis and the extent of the $y$ axis, we can easily use `animate` also for a function $T(z,t)$ (we just use $z$ as the name for the $x$ axis and $T$ as the name for the $y$ axis in the plot). Recall that it is important to fix the extent of the $y$ axis in a plot when we make animations, otherwise most plotting programs will automatically fit the extent of the axis to the current data, and the tickmarks on the $y$ axis will jump up and down during the movie. The result is a wrong visual impression of the function.

The names of the plot files must have a common stem appended with some frame number, and the frame number should have a fixed number of digits, such as 0001, 0002, etc. (if not, the sequence of the plot files

will not be correct when we specify the collection of files with an asterix for the frame numbers, e.g., as in `tmp*.png`). We therefore include an argument to `animate` for setting the name stem of the plot files. By default, the stem is `tmp_`, resulting in the filenames `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so forth. Other convenient arguments for the `animate` function are the initial time in the plot, the time lag $\Delta t$ between the plot frames, and the coordinates along the $x$ axis. The `animate` function then takes the form

```
def animate(tmax, dt, x, function, ymin, ymax, t0=0,
            xlabel='x', ylabel='y', hardcopy_stem='tmp_'):
    t = t0
    counter = 0
    while t <= tmax:
        y = function(x, t)
        plot(x, y,
             axis=[x[0], x[-1], ymin, ymax],
             title='time=%g' % t,
             xlabel=xlabel, ylabel=ylabel,
             hardcopy=hardcopy_stem + '%04d.png' % counter)
        t += dt
        counter += 1
```

The $T(z,t)$ function is easy to implement, but we need to decide whether the parameters $A$, $\omega$, $T_0$, and $k$ shall be arguments to the Python implementation of $T(z,t)$ or if they shall be global variables. Since the `animate` function expects that the function to be plotted has only two arguments, we must implement $T(z,t)$ as `T(z,t)` in Python and let the other parameters be global variables (Chapters 7.1.1 and 7.1.2 explain this problem in more detail and present a better implementation). The `T(z,t)` implementation then reads

```
def T(z, t):
    # T0, A, k, and omega are global variables
    a = sqrt(omega/(2*k))
    return T0 + A*exp(-a*z)*cos(omega*t - a*z)
```

Suppose we plot $T(z,t)$ at $n$ points for $z \in [0,D]$. We make such plots for $t \in [0,t_{\max}]$ with a time lag $\Delta t$ between the them. The frames in the movie are now made by

```
# set T0, A, k, omega, D, n, tmax, dt
z = linspace(0, D, n)
animate(tmax, dt, z, T, T0-A, T0+A, 0, 'z', 'T')
```

We have here set the extent of the $y$ axis in the plot as $[T_0-A, T_0+A]$, which is in accordance with the $T(z,t)$ function.

The call to `animate` above creates a set of files with names of the form `tmp_*.png`. The animation is then created by a call

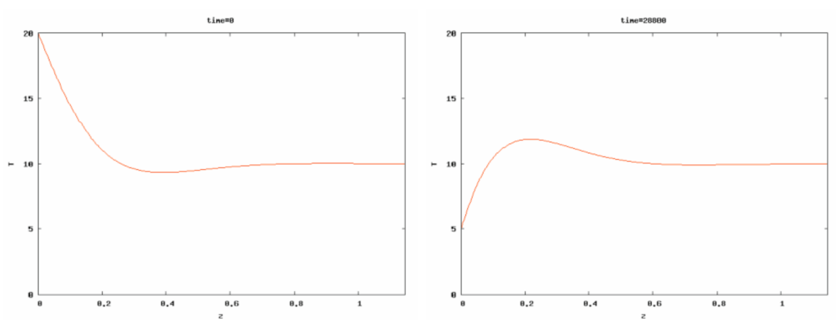
```
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')
```

It now remains to assign proper values to all the global variables in the program: `n`, `D`, `T0`, `A`, `omega`, `dt`, `tmax`, and `k`. The oscillation

period is 24 hours, and $\omega$ is related to the period $P$ of the cosine function by $\omega = 2\pi/P$ (realize that $\cos(t2\pi/P)$ has period $P$). We then express $P = 24$ h $= 24 \cdot 60 \cdot 60$ s and compute $\omega = 2\pi/P$. The total simulation time can be 3 periods, i.e., $t_{\max} = 3P$. The $T(z,t)$ function decreases exponentially with the depth $z$ so there is no point in having the maximum depth $D$ larger than the depth where $T$ is approximately zero, say 0.001. We have that $e^{-aD} = 0.001$ when $D = -a^{-1}\ln 0.001$, so we can use this estimate in the program. The proper initialization of all parameters can then be expressed as follows[14]:

```
k = 1E-6     # heat conduction coefficient (in m*m/s)
P = 24*60*60.# oscillation period of 24 h (in seconds)
omega = 2*pi/P
dt = P/24    # time lag: 1 h
tmax = 3*P   # 3 day/night simulation
T0 = 10      # mean surface temperature in Celsius
A = 10       # amplitude of the temperature variations in Celsius
a = sqrt(omega/(2*k))
D = -(1/a)*log(0.001) # max depth
n = 501      # no of points in the z direction
```

We encourage you to run the program `heatwave.py` to see the movie. The hardcopy of the movie is in the file `tmp_heatwave.gif`. Figure 4.13 displays two snapshots in time of the $T(z,t)$ function.




(a) (b)

**Fig. 4.13** Plot of the temperature $T(z,t)$ in the ground for two different $t$ values.

*Scaling.* In this example, as in many other scientific problems, it was easier to write the code than to assign proper physical values to the input parameters in the program. To learn about the physical process, here how heat propagates from the surface and down in the ground, it is often advantageous to scale the variables in the problem so that we work with dimensionless variables. Through the scaling procedure we normally end up with much fewer physical parameters which must be assigned values. Let us show how we can take advantage of scaling the present problem.

[14] Note that it is very important to use consistent units. Here we express all units in terms of meter, second, and Kelvin or Celsius.

Consider a variable $x$ in a problem with some dimension. The idea of scaling is to introduce a new variable $\bar{x} = x/x_c$, where $x_c$ is a *characteristic size* of $x$. Since $x$ and $x_c$ have the same dimension, the dimension cancels in $\bar{x}$ such that $\bar{x}$ is dimensionless. Choosing $x_c$ to be the expected maximum value of $x$, ensures that $\bar{x} \leq 1$, which is usually considered a good idea. That is, we try to have all dimensionless variables varying between zero and one. For example, we can introduce a dimensionless $z$ coordinate: $\bar{z} = z/D$, and now $\bar{z} \in [0, 1]$. Doing a proper scaling of a problem is challenging so for now it is sufficient to just follow the steps below - and not worry why we choose a certain scaling.

In the present problem we introduce these dimensionless variables:

$$
\begin{aligned}
\bar{z} &= z/D \\
\bar{T} &= \frac{T - T_0}{A} \\
\bar{t} &= \omega t
\end{aligned}
$$

We now insert $z = \bar{z}D$ and $t = \bar{t}/\omega$ in the expression for $T(z, t)$ and get

$$
T = T_0 + Ae^{-b\bar{z}} \cos(\bar{t} - b\bar{z}), \quad b = aD
$$

or

$$
\bar{T}(\bar{z}, \bar{t}) = \frac{T - T_0}{A} = e^{-b\bar{z}} \cos(\bar{t} - b\bar{z})\,.
$$

We see that $\bar{T}$ depends on only *one* dimensionless parameter $b$ in addition to the independent dimensionless variables $\bar{z}$ and $\bar{t}$. It is common practice at this stage of the scaling to just drop the bars and write

$$
T(z, t) = e^{-bz} \cos(t - bz)\,. \tag{4.14}
$$

This function is much simpler to plot than the one with lots of physical parameters, because now we know that $T$ varies between $-1$ and $1$, $t$ varies between $0$ and $2\pi$ for one period, and $z$ varies between 0 and 1. The scaled temperature has only one "free" parameter $b$. That is, the shape of the graph is completely determined by $b$.

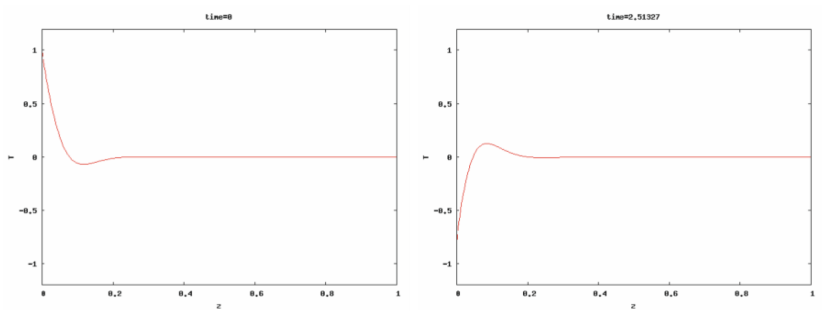
In our previous movie example, we used specific values for $D$, $\omega$, and $k$, which then implies a certain $b = D\sqrt{\omega/(2k)}$ ($\approx 6.9$). However, we can now run different $b$ values and see the effect on the heat propagation. Different $b$ values will in our problems imply different periods of the surface temperature variation and/or different heat conduction values in the ground's composition of rocks. Note that doubling $\omega$ and $k$ leaves the same $b$ – it is only the fraction $\omega/k$ that influences the value of $b$.

In a main program we can read $b$ from the command line and make the movie:

```
b = float(sys.argv[1])
n = 401
z = linspace(0, 1, n)
animate(3*2*pi, 0.05*2*pi, z, T, -1.2, 1.2, 0, 'z', 'T')
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')
```

Running the program, found as the file `heatwave_scaled.py`, for different $b$ values shows that $b$ governs how deep the temperature variations on the surface $z = 0$ penetrate. A large $b$ makes the temperature changes confined to a thin layer close to the surface (see Figure 4.14 for $b = 20$), while a small $b$ leads to temperature variations also deep down in the ground (see Figure 4.15 for $b = 2$).

We can understand the results from a physical perspective. Think of increasing $\omega$, which means reducing the oscillation period so we get a more rapid temperature variation. To preserve the value of $b$ we must increase $k$ by the same factor. Since a large $k$ means that heat quickly spreads down in the ground, and a small $k$ implies the opposite, we see that more rapid variations at the surface requires a larger $k$ to more quickly conduct the variations down in the ground. Similarly, slow temperature variations on the surface can penetrate deep in the ground even if the ground's ability to conduct ($k$) is low.




(a) (b)

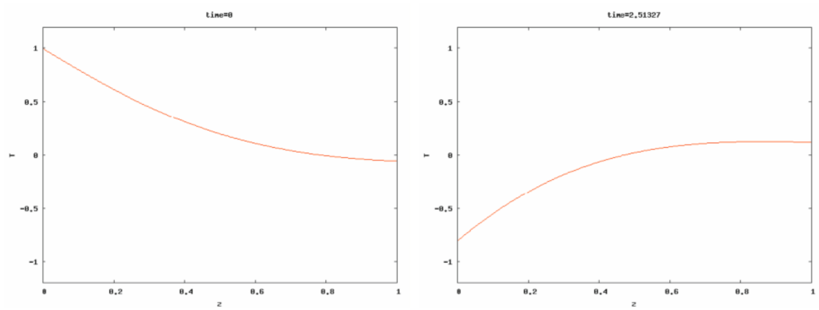
**Fig. 4.14** Plot of the dimensionless temperature $T(z, t)$ in the ground for two different $t$ values and $b = 20$.

## 4.8 Exercises

**Exercise 4.1.** *Fill lists with function values.*

A function with many applications in science is defined as

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} . \tag{4.15}$$




(a) (b)

**Fig. 4.15** Plot of the dimensionless temperature $T(z,t)$ in the ground for two different $t$ values and $b=2$.

Fill lists `xlist` and `hlist` with $x$ and $h(x)$ values for uniformly spaced $x$ coordinates in $[-4,4]$. You may adapt the example in Chapter 4.2.1. Name of program file: `fill_lists.py`. ◇

**Exercise 4.2.** *Fill arrays; loop version.*

The aim is to fill two arrays `x` and `h` with $x$ and $h(x)$ values, where $h(x)$ is defined in (4.15). Let the $x$ values be uniformly spaced in $[-4,4]$. Create two arrays of zeros and fill both arrays with values inside a loop. Name of program file: `fill_arrays_loop.py`. ◇

**Exercise 4.3.** *Fill arrays; vectorized version.*

Vectorize the code in Exercise 4.2 by creating the $x$ values using the `linspace` function and by evaluating $h(x)$ for an array argument. Name of program file: `fill_arrays_vectorized.py`. ◇

**Exercise 4.4.** *Apply a function to a vector.*

Given a vector $v=(2,3,-1)$ and a function $f(x)=x^3+xe^x+1$, apply $f$ to each element in $v$. Then calculate $f(v)$ as $v^3+v*e^v+1$ using the vector computing rules. Show that the two results are equal. ◇

**Exercise 4.5.** *Simulate by hand a vectorized expression.*

Suppose `x` and `t` are two arrays of the same length, entering a vectorized expression

```
y = cos(sin(x)) + exp(1/t)
```

If `x` holds two elements, 0 and 2, and `t` holds the elements 1 and 1.5, calculate by hand (using a calculator) the `y` array. Thereafter, write a program that mimics the series of computations you did by hand (typically a sequence of operations of the kind we listed on page 174 – use explicit loops, but at the end you can use Numerical Python functionality to check the results). Name of program file: `simulate_vector_computing.py`. ◇

**Exercise 4.6.** *Demonstrate array slicing.*

Create an array `w` with values $0, 0.1, 0.2, \ldots, 2$ using `linspace`. Write out `w[:]`, `w[:-2]`, `w[::5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed. Name of program file: `slicing.py`. ◇

**Exercise 4.7.** *Plot the formula (1.1).*

Make a plot of the function $y(t)$ in (1.1) on page 1 for $v_0 = 10$ and $t \in [0, 2v_0/g]$. The label on the $x$ axis should be 'time'.

If you use Easyviz with the Gnuplot backend on Windows machines, you need a `raw_input()` call at the end of the program such that the program halts until Gnuplot is finished with the plot (simply press Return to finish the program). See also Appendix E.1.3.

Name of program file: `plot_ball1.py`. ◇

**Exercise 4.8.** *Plot the formula (1.1) for several $v_0$ values.*

Make a program that reads a set of $v_0$ values from the command line and plots the curve (1.1) for the different $v_0$ values (in the same figure). Let the $t$ coordinates go from 0 to $2v_0/g$ for each curve, which implies that you need a different vector of $t$ coordinates for each curve. Name of program file: `plot_ball2.py`. ◇

**Exercise 4.9.** *Plot exact and inexact Fahrenheit–Celsius formulas.*

Exercise 2.20 introduces a simple rule to quickly compute the Celsius temperature from the Fahreheit degrees: $C = (F - 30)/2$. Compare this curve against the exact curve $C = (F - 32)5/9$ in a plot. Let $F$ vary between $-20$ and 120. Name of program file: `f2c_shortcut_plot.py`. ◇

**Exercise 4.10.** *Plot the trajectory of a ball.*

The formula for the trajectory of a ball is given in (1.5) on page 38. In a program, first read the input data $y_0$, $\theta$, and $v_0$ from the command line. Then compute where the ball hits the ground, i.e., the value $x_g$ for which $f(x_g) = 0$. Plot the trajectory $y = f(x)$ for $x \in [0, x_g]$, using the same scale on the $x$ and $y$ axes such that we get a visually correct view of the trajectory. Name of program file: `plot_trajectory.py`. ◇

**Exercise 4.11.** *Plot a wave packet.*

The function

$$f(x,t) = e^{-(x-3t)^2} \sin(3\pi(x-t)) \tag{4.16}$$

describes for a fixed value of $t$ a wave localized in space. Make a program that visualizes this function as a function of $x$ on the interval $[-4, 4]$ when $t = 0$. Name of program file: `plot_wavepacket.py`. ◇

**Exercise 4.12.** *Use pyreport in Exer. 4.11.*

Use `pyreport` (see Chapter 1.8) in Exercise 4.11 to generate a nice report in HTML and PDF format. To get the plot inserted in the

report, you must call `show()` after the `plot` instruction. You also need to use the version of `pyreport` that comes with SciTools (that version is automatically installed when you install SciTools). Name of program file: `pyreport_wavepacket.py`. ⋄

**Exercise 4.13.** *Judge a plot.*

Assume you have the following program for plotting a parabola:

```
x = linspace(0, 2, 20)
y = x*(2 - x)
plot(x, y)
```

Then you switch to the function $\cos(18\pi x)$ by altering the computation of `y` to `y = cos(18*pi*x)`. Judge the resulting plot. Is it correct? Display the $\cos(18\pi x)$ function with 1000 points in the same plot. Name of program file: `judge_plot.py`. ⋄

**Exercise 4.14.** *Plot the viscosity of water.*

The viscosity of water, $\mu$, varies with the temperature $T$ (in Kelvin) according to

$$\mu(T) = A \cdot 10^{B/(T-C)}, \tag{4.17}$$

where $A = 2.414 \cdot 10^{-5}$ Ns/m$^2$, $B = 247.8$ K, and $C = 140$ K. Plot $\mu(T)$ for $T$ between 0 and 100 degrees Celsius. ⋄

**Exercise 4.15.** *Explore a function graphically.*

The wave speed of water surface waves, $c$, depends on the length of the wave, $\lambda$. The following formula relates $c$ to $\lambda$:

$$c(\lambda) = \sqrt{\frac{g\lambda}{2\pi}\left(1 + s\frac{4\pi^2}{\rho g \lambda^2}\right)\tanh\left(\frac{2\pi h}{\lambda}\right)}, \tag{4.18}$$

where $g$ is the acceleration of gravity, $s$ is the surface tension between water and air ($7.9 \cdot 10^{-4}$ N/cm) , $\rho$ is the density of water (can be taken as 1 kg/cm$^3$), and $h$ is the water depth. Let us fix $h$ at 50 m. First make a plot of $c(\lambda)$ for small $\lambda$ (1 mm to 10 cm). Then make a plot $c(\lambda)$ for larger $\lambda$ (1 m to 2 km). Name of program file: `water_wave_velocity.py`.
⋄

**Exercise 4.16.** *Plot Taylor polynomial approximations to* $\sin x$.

The sine function can be approximated by a polynomial according to the following formula:

$$\sin x \approx S(x; n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!} \,. \tag{4.19}$$

The expression $(2j + 1)!$ is the factorial (see Exercise 2.33). The error in the approximation $S(x; n)$ decreases as $n$ increases and in the limit

we have that $\sin x = \lim_{n\to\infty} S(x; n)$. The purpose of this exercise is to visualize the quality of various approximations $S(x; n)$ as $n$ increases.

The first part of the exercise is to write a Python function `S(x, n)` that computes $S(x; n)$. Use a straightforward approach where you compute each term as it stands in the formula, i.e., $(-1)^j x^{2j+1}$ divided by the factorial $(2j + 1)!$. (We remark that Exercise 5.16 outlines a much more efficient computation of the terms in the series.)

The next part of the exercise is to plot $\sin x$ on $[0, 4\pi]$ together with the approximations $S(x; 1)$, $S(x; 2)$, $S(x; 3)$, $S(x; 6)$, and $S(x; 12)$. Name of program file: `plot_Taylor_sin.py`. ◇

**Exercise 4.17.** *Animate a wave packet.*

Display an animation of the function $f(x, t)$ in Exercise 4.11 by plotting $f$ as a function of $x$ on $[-6, 6]$ for a set of $t$ values in $[-1, 1]$. Also make an animated GIF file. A suitable resolution can be 1000 intervals (1001 points) along the $x$ axis, 60 intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use the recipe in Chapter 4.3.7 and remember to remove the family of old plot files in the beginning of the program.

You will see that $f(x, t)$ models waves that are moving to the right (when $x$ is a space coordinate and $t$ is time). The velocities of the individual waves and the packet are different, demonstrating an important case in physics when the phase velocity of waves is different from the group velocity. This effect is visible for water surface waves, particularly those generated by a boat. Name of program file: `plot_wavepacket_movie.py`. ◇

**Exercise 4.18.** *Animate the evolution of Taylor polynomials.*

A general series approximation (to a function) can be written as

$$S(x; M, N) = \sum_{k=M}^{N} f_k(x) .$$

For example, the Taylor polynomial for $e^x$ equals $S(x)$ with $f_k(x) = x^k/k!$. The purpose of the exercise is to make a movie of how $S(x)$ develops (and hopefully improves as an approximation) as we add terms in the sum. That is, the frames in the movie correspond to plots of $S(x; M, M + 1)$, $S(x; M, M + 2)$, ..., $S(x; M, N)$.

Make a function

```
animate_series(fk, M, N, xmin, xmax, ymin, ymax, n, exact)
```

for creating such animations. The argument `fk` holds a Python function implementing the term $f_k(x)$ in the sum, `M` and `N` are the summation limits, the next arguments are the minimum and maximum $x$ and $y$ values in the plot, `n` is the number of $x$ points in the curves to be plotted, and `exact` holds the function that $S(x)$ aims at approximating.

The `animate_series` function must accumulate the $f_k(x)$ in a variable $s$, and for each $k$ value, $s$ is plotted against $x$ together with a curve reflecting the exact function. Each plot must be saved in a file, say with names `tmp_0000.png`, `tmp_0001.png`, and so on (these filenames can be generated by `tmp_%04d.png`, using an appropriate counter). Use the `movie` function to combine all the plot files into a movie in a desired movie format.

In the beginning of the `animate_series` it is necessary to remove all old plot files of the form `tmp_*.png`. This can be done by the `glob` module and the `os.remove` function as exemplified in Chapter 4.3.7 and in Appendix E.4 (page 677).

Test the `animate_series` function in the two cases:

1. The Taylor series for $\sin x$, where $f_k(x) = (-1)^k x^{2k+1}/(2k+1)!$, and $x \in [0, 13\pi]$, $M = 0$, $N = 40$, $y \in [-2, 2]$.
2. The Taylor series for $e^{-x}$, where $f_k(x) = (-x)^k/k!$, and $x \in [0, 15]$, $M = 0$, $N = 30$, $y \in [-0.5, 1.4]$.

Name of program file: `animate_Taylor_series.py`. ◇

**Exercise 4.19.** *Plot the velocity profile for pipeflow.*

A fluid that flows through a (very long) pipe has zero velocity on the pipe wall and a maximum velocity along the centerline of the pipe. The velocity $v$ varies through the pipe cross section according to the following formula:

$$v(r) = \left(\frac{\beta}{2\mu_0}\right)^{1/n} \frac{n}{n+1}\left(R^{1+1/n} - r^{1+1/n}\right), \tag{4.20}$$

where $R$ is the radius of the pipe, $\beta$ is the pressure gradient (the force that drives the flow through the pipe), $\mu_0$ is a viscosity coefficient (small for air, larger for water and even larger for toothpaste), $n$ is a real (!) number reflecting the viscous properties of the fluid ($n = 1$ for water and air, $n < 1$ for many modern plastic materials), and $r$ is a radial coordinate that measures the distance from the centerline ($r = 0$ is the centerline, $r = R$ is the pipe wall).

Make a function that evaluates $v(r)$. Plot $v(r)$ as a function of $r \in [0, R]$, with $R = 1$, $\beta = 0.02$, $\mu = 0.02$, and $n = 0.1$. Thereafter, make an animation of how the $v(r)$ curves varies as $n$ goes from 1 and down to 0.01. Because the maximum value of $v(r)$ decreases rapidly as $n$ decreases, each curve can be normalized by its $v(0)$ value such that the maximum value is always unity. Name of program file: `plot_velocity_pipeflow.py`. ◇

**Exercise 4.20.** *Plot the approximate function from Exer. 1.13.*

First make a Python function `S(t, n)` that evaluates $S(t; n)$ defined in Exercise 1.13. Plot $S(t; 1)$, $S(t; 3)$, $S(t; 20)$, $S(t; 200)$, and the exact $f(t)$ function in the same plot.

The resulting plot shows how a step-like function can be approximated by a sum of sine functions of increasing frequency. Representation of functions as a sum of sines and cosines is an important mathematical concept, known as Fourier series, and has a wide range of applications. Name of program file: `plot_compare_func_sum.py`. ◇

**Exercise 4.21.** *Plot functions from the command line.*

For quickly get a plot a function $f(x)$ for $x \in [x_{\min}, x_{\max}]$ it could be nice to a have a program that takes the minimum amount of information from the command line and produces a plot on the screen and a hardcopy `tmp.eps`. The usage of the program goes as follows:

Terminal
```
plotf.py "f(x)" xmin xmax
```

A specific example is

Terminal
```
plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

Hint: Make $x$ coordinates from the 2nd and 3rd command-line arguments and then use `eval` (or `StringFunction` from Chapters 3.1.4 and 4.4.3) on the first 1st argument. Try to write as short program as possible (we leave it to Exercise 4.22 to test for valid input). Name of program file: `plotf_v1.py`. ◇

**Exercise 4.22.** *Improve the program from Exericse 4.21.*

Equip the program from Exericse 4.21 with tests on valid input on the command line. Also allow an optional 4th command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Name of program file: `plotf.py`. ◇

**Exercise 4.23.** *Demonstrate energy concepts from physics.*

The vertical position $y(t)$ of a ball thrown upward is described by (1.1) on page 1. Two important physical quantities in this context are the potential energy, obtained by from doing work against gravity, and the kinetic energy, arising from motion. The potential energy is defined as $P = mgy$, where $m$ is the mass of the ball. The kinetic energy is defined as $K = \frac{1}{2}mv^2$, where $v$ is the velocity of the ball, related to $y$ by $v(t) = y'(t)$. Plot $P(t)$ and $K(t)$ in the same plot, along with their sum $P + K$. Let $t \in \frac{2v_0}{g}$. Read $m$ and $v_0$ from the command line. Run the program with various choices of $m$ and $v_0$ and observe that $P + K$ is always constant in this motion. In fact, it turns out that $P + K$ is constant for a large class of motions, and this is a very important result in physics. Name of program file: `energy_physics.py`. ◇

**Exercise 4.24.** *Plot a w-like function.*

Define mathematically a function that looks like the 'w' character. Plot this function. Name of program file: `plot_w.py`. ◇

**Exercise 4.25.** *Plot a smoothed "hat" function.*

The "hat" function $N(x)$ defined by (2.5) on page 89 has a discontinuity in the derivative at $x = 1$. Suppose we want to "round" this function such that it looks smooth around $x = 1$. To this end, replace the straight lines in the vicinity of $x = 1$ by a (small) cubic curve

$$y = a(x-1)^3 + b(x-1)^2 + c(x-1) + d,$$

for $x \in [1-\epsilon, 1+\epsilon]$, where $a$, $b$, $c$, and $d$ are parameters that must be adjusted in order for the cubic curve to match the value and the derivative of the function $N(x)$. The new rounded functions has the specification

$$\tilde{N}(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x < 1-\epsilon \\ a_1(x-1)^3 + b(x-1) + c(x-1) + d_1, & 1-\epsilon \le x < 1, \\ a_2(x-1)^3 + b(x-1) + c(x-1) + d_2, & 1 \le x < 1+\epsilon, \\ 2-x, & 1+\epsilon \le x < 2 \\ 0, & x \ge 2 \end{cases} \tag{4.21}$$

with $a_1 = \frac{1}{3}\epsilon^{-2}$, $a_2 = -a_1$, $d_1 = 1-\epsilon+a_1\epsilon^3$, $d_2 = 1-\epsilon-a_2\epsilon^3$, and $b = c = 0$. Plot this function. (Hint: Be careful with the choice of $x$ coordinates!) Name of program file: `plot_hat.py`. ◇

**Exercise 4.26.** *Experience overflow in a function.*

When object (ball, car, airplane) moves through the air, there is a very, very thin layer of air close to the object's surface where the air velocity varies dramatically[15], from the same value as the velocity of the object at the object's surface to zero a few centimeters away. The change in velocity is quite abrupt and can modeled by the functiion

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}},$$

where $x = 1$ is the object's surface and $x = 0$ is some distance "far" away where one cannot notice any wind velocity $v$ because of the passing object ($v = 0$). The vind velocity coincides with the velocity of the object at $x = 1$, here set to $v = 1$. The parameter $\mu$ reflects the viscous behavior of air, which is very small, so typically $\mu = 10^{-6}$. With this relevant physical value of $\mu$, it quickly becomes difficult to calculate $v(x)$ on a computer.

Make a function `v(x, mu=1E-6, exp=math.exp)` for calculating the formula of $v$ using `exp` as a possibly user-given exponentional function. Let the `v` function return the nominator and denominator in the formula as well as the fraction (result). Call the `v` function for various `x`

[15] This layer is called a *boundary layer*. The physics in the boundary layer is very important for air resistance and cooling/heating of objects.

values between 0 and 1 in a `for` loop, let `mu` be `1E-3`, and have an inner `for` loop over two different `exp` functions: `math.exp` and `numpy.exp`. The output will demonstrate how the denominator is subject to overflow and how difficult it is to calculate this function on a computer.

Also plot $v(x)$ for $\mu = 1, 0.01, 0.001$ on $[0, 1]$ using 10,000 points to see what the function looks like. Name of program file: `boundary_layer_func1.py`. ◇

**Exercise 4.27.** *Experience less overflow in a function.*

In the program from Exercise 4.26, convert `x` and `eps` to a higher presicion representation of real numbers, with the aid of the NumPy type `float96`:

```
import numpy
x = numpy.float96(x); mu = numpy.float96(e)
```

Call the `v` function with these type of variables observe how much "better" results we get with `float96` compared the standard `float` value (which is `float64` – the number reflects the number of bits in the machine's representation of a real number). Also call the `v` function with `x` and `mu` as `float32` variables and report how the function now behaves. Name of program file: `boundary_layer_func2.py`. ◇

**Exercise 4.28.** *Extend Exer. 4.4 to a rank 2 array.*

Let $A$ be a two-dimensional array with two indices:

$$\begin{bmatrix} 0 & 12 & -1 \\ -1 & -1 & -1 \\ 11 & 5 & 5 \end{bmatrix}$$

Apply the function $f$ from Exercise 4.4 to each element in $A$. Thereafter, calculate the result of the array expression $A**3 + A*e^A + 1$ and demonstrate that the end result of the two methods are the same. ◇

**Exercise 4.29.** *Explain why array computations fail.*

The following loop computes the array `y` from `x`:

```
>>> x = linspace(0, 1, 3)
>>> y = zeros(len(x))
>>> for i in range(len(x)):
...     y[i] = x[i] + 4
```

However, the alternative loop

```
>>> for xi, yi in zip(x, y):
...     yi = xi + 5
```

leaves `y` unchanged. Why? Explain in detail what happens in each pass of this loop and write down the contents of `xi`, `yi`, `x`, and `y` as the loop progresses. ◇

# 5 Sequences and Difference Equations

From mathematics you probably know the concept of a *sequence*, which is nothing but a collection of numbers with a specific order. A general sequence is written as

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_n, \ldots,$$

One example is the sequence of all odd numbers:

$$1, 3, 5, 7, \ldots, 2n+1, \ldots$$

For this sequence we have an explicit formula for the $n$-th term: $2n+1$, and $n$ takes on the values 0, 1, 2, .... We can write this sequence more compactly as $(x_n)_{n=0}^{\infty}$ with $x_n = 2n+1$. Other examples of infinite sequences from mathematics are

$$1,\ 4,\ 9,\ 16,\ 25,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = (n+1)^2, \tag{5.1}$$

$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \frac{1}{n+1}\,. \tag{5.2}$$

The former sequences are infinite, because they are generated from all integers $\geq 0$ and there are infinitely many such integers. Nevertheless, most sequences from real life applications are finite. If you put an amount $x_0$ of money in a bank, you will get an interest rate and therefore have an amount $x_1$ after one year, $x_2$ after two years, and $x_N$ after $N$ years. This process results in a finite sequence of amounts

$$x_0, x_1, x_2, \ldots, x_N, \quad (x_n)_{n=0}^{N}\,.$$

Usually we are interested in quite small $N$ values (typically $N \leq 20-30$). Anyway, the life of the bank is finite, so the sequence definitely has an end.

For some sequences it is not so easy to set up a general formula for the $n$-th term. Instead, it is easier to express a relation between two or more consecutive elements. One example where we can do both things is the sequence of odd numbers. This sequence can alternatively be generated by the formula

$$x_{n+1} = x_n + 2\,. \tag{5.3}$$

To start the sequence, we need an *initial condition* where the value of the first element is specified:

$$x_0 = 1\,.$$

Relations like (5.3) between consecutive elements in a sequence is called recurrence relations or *difference equations*. Solving a difference equation can be quite challenging in mathematics, but it is almost trivial to solve it on a computer. That is why difference equations are so well suited for computer programming, and the present chapter is devoted to this topic.

The program examples regarding difference equations are found in the folder `src/diffeq`.

## 5.1 Mathematical Models Based on Difference Equations

The objective of science is to understand complex phenomena. The phenomenon under consideration may be a part of nature, a group of social individuals, the traffic situation in Los Angeles, and so forth. The reason for addressing something in a scientific manner is that it appears to be complex and hard to comprehend. A common scientific approach to gain understanding is to create a model of the phenomenon, and discuss the properties of the model instead of the phenomenon. The basic idea is that the model is easier to understand, but still complex enough to preserve the basic features of the problem at hand[1]. Modeling is, indeed, a general idea with applications far beyond science. Suppose, for instance, that you want to invite a friend to your home for the first time. To assist your friend, you may send a map of your neighborhood. Such a map is a model: It exposes the most important landmarks and leave out billions of details that your friend can do very well without. This is the essence of modeling: A good model should be as simple as possible, but still rich enough to include the important structures you are looking for[2].

[1] “Essentially, all models are wrong, but some are useful.” –George E. P. Box, statistician, 1919-.

[2] ”Everything should be made as simple as possible, but not simpler.” –Albert Einstein, physicist, 1879-1955.

Certainly, the tools we apply to model a certain phenomenon differ a lot in various scientific disciplines. In the natural sciences, mathematics has gained a unique position as the key tool for formulating models. To establish a model, you need to understand the problem at hand and describe it with mathematics. Usually, this process results in a set of equations, i.e., the model consists of equations that must be solved in order to see how realistically the model describes a phenomenon. Difference equations represent one of the simplest yet most effective type of equations arising in mathematical models. The mathematics is simple and the programming is simple, thereby allowing us to focus more on the modeling part. Below we will derive and solve difference equations for diverse applications.

### 5.1.1 Interest Rates

Our first difference equation model concerns how much money an initial amount $x_0$ will grow to after $n$ years in a bank with annual interest rate $p$. You learned in school the formula

$$x_n = x_0 \left(1 + \frac{p}{100}\right)^n . \tag{5.4}$$

Unfortunately, this formula arises after some limiting assumptions, like that of a constant interest rate over all the $n$ years. Moreover, the formula only gives us the amount after each year, not after some months or days. It is much easier to compute with interest rates if we set up a more fundamental model in terms of a difference equation and then solve this equation on a computer.

The fundamental model for interest rates is that an amount $x_{n-1}$ at some point of time $t_{n-1}$ increases its value with $p$ percent to an amount $x_n$ at a new point of time $t_n$:

$$x_n = x_{n-1} + \frac{p}{100} x_{n-1} . \tag{5.5}$$

If $n$ counts years, $p$ is the annual interest rate, and if $p$ is constant, we can with some arithmetics derive the following solution to (5.5):

$$x_n = \left(1 + \frac{p}{100}\right) x_{n-1} = \left(1 + \frac{p}{100}\right)^2 x_{n-2} = \ldots = \left(1 + \frac{p}{100}\right)^n x_0 .$$

Instead of first deriving a formula for $x_n$ and then program this formula, we may attack the fundamental model (5.5) in a program (`growth_years.py`) and compute $x_1$, $x_2$, and so on in a loop:

```
from scitools.std import *
x0 = 100                      # initial amount
p = 5                         # interest rate
N = 4                         # number of years
index_set = range(N+1)
```

```
x = zeros(len(index_set))

# solution:
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

The output of `x` is

```
[ 100.       105.       110.25      115.7625    121.550625]
```

Programmers of mathematical software who are trained in making programs more efficient, will notice that it is not necessary to store all the $x_n$ values in an array or use a list with all the indices $0, 1, \ldots, N$. Just one integer for the index and two floats for $x_n$ and $x_{n-1}$ are strictly necessary. This can save quite some memory for large values of $N$. Exercise 5.5 asks you to develop such a memory-efficient program.

Suppose now that we are interested in computing the growth of money after $N$ days instead. The interest rate per day is taken as $r = p/D$ if $p$ is the annual interest rate and $D$ is the number of days in a year. The fundamental model is the same, but now $n$ counts days and $p$ is replaced by $r$:

$$x_n = x_{n-1} + \frac{r}{100} x_{n-1}\,. \tag{5.6}$$

A common method in international business is to choose $D = 360$, yet let $n$ count the exact number of days between two dates (see footnote on page 142). Python has a module `datetime` for convenient calculations with dates and times. To find the number of days between two dates, we perform the following operations:

```
>>> import datetime
>>> date1 = datetime.date(2007, 8, 3)  # Aug 3, 2007
>>> date2 = datetime.date(2008, 8, 4)  # Aug 4, 2008
>>> diff = date2 - date1
>>> print diff.days
367
```

We can modify the previous program to compute with days instead of years:

```
from scitools.std import *
x0 = 100                          # initial amount
p = 5                             # annual interest rate
r = p/360.0                       # daily interest rate
import datetime
date1 = datetime.date(2007, 8, 3)
date2 = datetime.date(2011, 8, 3)
diff = date2 - date1
N = diff.days
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
```

```
for n in index_set[1:]:
    x[n] = x[n-1] + (r/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='days', ylabel='amount')
```

Running this program, called `growth_days.py`, prints out 122.5 as the final amount.

It is quite easy to adjust the formula (5.4) to the case where the interest is added every day instead of every year. However, the strength of the model (5.6) and the associated program `growth_days.py` becomes apparent when $r$ varies in time – and this is what happens in real life. In the model we can just write $r(n)$ to explicitly indicate the dependence upon time. The corresponding time-dependent annual interest rate is what is normally specified, and $p(n)$ is usually a piecewise constant function (the interest rate is changed at some specific dates and remains constant between these days). The construction of a corresponding array `p` in a program, given the dates when $p$ changes, can be a bit tricky since we need to compute the number of days between the dates of changes and index `p` properly. We do not dive into these details now, but readers who want to compute `p` and who is ready for some extra brain training and index puzzling can attack Exercise 5.11. For now we assume that an array `p` holds the time-dependent annual interest rates for each day in the total time period of interest. The `growth_days.py` program then needs a slight modification, typically,

```
p = zeros(len(index_set))
# set up p (might be challenging!)
r = p/360.0                         # daily interest rate
...
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]
```

For the very simple (and not-so-relevant) case where $p$ grows linearly (i.e., daily changes) from 4 to 6 percent over the period of interest, we have made a complete program in the file `growth_days_timedep.py`. You can compare a simulation with linearly varying $p$ between 4 and 6 and a simulation using the average $p$ value 5 throughout the whole time interval.

A difference equation with $r(n)$ is quite difficult to solve mathematically, but the $n$-dependence in $r$ is easy to deal with in the computerized solution approach.

### 5.1.2 The Factorial as a Difference Equation

The difference equation

$$x_n = nx_{n-1}, \quad x_0 = 1 \tag{5.7}$$

can quickly be solved recursively:

$$\begin{aligned}x_n &= nx_{n-1}\\&= n(n-1)x_{n-2}\\&= n(n-1)(n-2)x_{n-3}\\&= n(n-1)(n-2)\cdots 1\,.\end{aligned}$$

The result $x_n$ is nothing but the factorial of $n$, denoted as $n!$ (cf. Exercise 2.33). Equation (5.7) then gives a standard recipe to compute $n!$.

### 5.1.3 Fibonacci Numbers

Every textbook with some material on sequences usually presents a difference equation for generating the famous Fibonacci numbers[3]:

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1,\ x_1 = 1,\ n = 2, 3, \ldots \tag{5.8}$$

This equation has a relation between three elements in the sequence, not only two as in the other examples we have seen. We say that this is a difference equation of second order, while the previous examples involving two $n$ levels are said to be difference equations of first order. The precise characterization of (5.8) is a homogeneous difference equation of second order. Such classification is not important when computing the solution in a program, but for mathematical solution methods by pen and paper, the classification helps to determine which mathematical technique to use to solve the problem.

A straightforward program for generating Fibonacci numbers takes the form (`fibonacci1.py`):

```
import sys
from numpy import zeros
N = int(sys.argv[1])
x = zeros(N+1, int)
x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
    print n, x[n]
```

Since $x_n$ is an infinite sequence we could try to run the program for very large $N$. This causes two problems: The storage requirements of the `x` array may become too large for the computer, but long before this happens, $x_n$ grows in size far beyond the largest integer that can be represented by `int` elements in arrays (the problem appears already for $N = 50$). A possibility is to use array elements of type `int64`, which allows computation of twice as many numbers as with standard `int` elements (see the program `fibonacci1_int64.py`). A better solution is to use `float` elements in the `x` array, despite the fact that the numbers

[3] Fibonacci arrived at this equation when modelling rabbit populations.

$x_n$ are integers. With `float96` elements we can compute up to $N = 23600$ (see the program `fibinacci1_float.py`).

The best solution goes as follows. We observe, as mentioned after the `growth_years.py` program and also explained in Exercise 5.5, that we need only three variables to generate the sequence. We can therefore work with just three standard `int` variables in Python:

```
import sys
N = int(sys.argv[1])
xnm1 = 1
xnm2 = 1
n = 2
while n <= N:
    xn = xnm1 + xnm2
    print 'x_%d = %d' % (n, xn)
    xnm2 = xnm1
    xnm1 = xn
    n += 1
```

Here `xnm1` denotes $x_{n-1}$ and `xnm2` denotes $x_{n-2}$. To prepare for the next pass in the loop, we must shuffle the `xnm1` down to `xnm2` and store the new $x_n$ value in `xnm1`. The nice thing with `int` objects in Python (contrary to `int` elements in NumPy arrays) is that they can hold integers of arbitrary size[4]. We may try a run with `N` set to 250:

```
x_2 = 2
x_3 = 3
x_4 = 5
x_5 = 8
x_6 = 13
x_7 = 21
x_8 = 34
x_9 = 55
x_10 = 89
x_11 = 144
x_12 = 233
x_13 = 377
x_14 = 610
x_15 = 987
x_16 = 1597
...
x_249 = 7896325826131730509282738943634332893686268675876375
x_250 = 12776523572924732586037033894655031898659556447352249
```

In mathematics courses you learn how to derive a formula for the $n$-th term in a Fibonacci sequence. This derivation is much more complicated than writing a simple program to generate the sequence, but there is a lot of interesting mathematics both in the derivation and the resulting formula!

### 5.1.4 Growth of a Population

Let $x_{n-1}$ be the number of individuals in a population at time $t_{n-1}$. The population can consists of humans, animals, cells, or whatever objects where the number of births and deaths is proportional to the number of individuals. Between time levels $t_{n-1}$ and $t_n$, $bx_n$ individuals are born,

[4] Note that `int` variables in other computer languages normally has a size limitation like `int` elements in NumPy arrays.

and $dx_n$ individuals die, where $b$ and $d$ are constants. The net growth of the population is then $(b-d)x_n$. Introducing $r=(b-d)100$ for the net growth factor measured in percent, the new number of individuals become

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1}\,. \tag{5.9}$$

This is the same difference equation as (5.5). It models growth of populations quite well as long as there are optimal growing conditions for each individual. If not, one can adjust the model as explained in Chapter 5.1.5.

To solve (5.9) we need to start out with a known size $x_0$ of the population. The $b$ and $d$ parameters depend on the time difference $t_n - t_{n-1}$, i.e., the values of $b$ and $d$ are smaller if $n$ counts years than if $n$ counts generations.

### 5.1.5 Logistic Growth

The model (5.9) for the growth of a population leads to exponential increase in the number of individuals as implied by the solution 5.4. The size of the population increases faster and faster as time $n$ increases, and $x_n \to \infty$ when $n \to \infty$. In real life, however, there is an upper limit $M$ of the number of individuals that can exist in the environment at the same time. Lack of space and food, competition between individuals, predators, and spreading of contagious diseases are examples on factors that limit the growth. The number $M$ is usually called the *carrying capacity* of the environment, the maximum population which is sustainable over time. With limited growth, the growth factor $r$ must depend on time:

$$x_n = x_{n-1} + \frac{r(n-1)}{100}x_{n-1}\,. \tag{5.10}$$

In the beginning of the growth process, there is enough resources and the growth is exponential, but as $x_n$ approaches $M$, the growth stops and $r$ must tend to zero. A simple function $r(n)$ with these properties is

$$r(n) = \varrho\left(1 - \frac{x_n}{M}\right)\,. \tag{5.11}$$

For small $n$, $x_n \ll M$ and $r(n) \approx \varrho$, which is the growth rate with unlimited resources. As $n \to M$, $r(n) \to 0$ as we want. The model (5.11) is used for *logistic growth*. The corresponding *logistic difference equation* becomes

$$x_n = x_{n-1} + \frac{\varrho}{100}x_{n-1}\left(1 - \frac{x_{n-1}}{M}\right)\,. \tag{5.12}$$

Below is a program (`growth_logistic.py`) for simulating $N=200$ time intervals in a case where we start with $x_0 = 100$ individuals, a carrying

capacity of $M = 500$, and initial growth of $\varrho = 4$ percent in each time interval:

```
from scitools.std import *
x0 = 100                # initial amount of individuals
M = 500                 # carrying capacity
rho = 4                 # initial growth rate in percent
N = 200                 # number of time intervals
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(M))
print x
plot(index_set, x, 'r', xlabel='time units',
     ylabel='no of individuals', hardcopy='tmp.eps')
```

Figure 5.1 shows how the population stabilizes, i.e., that $x_n$ approaches $M$ as $N$ becomes large (of the same magnitude as $M$).

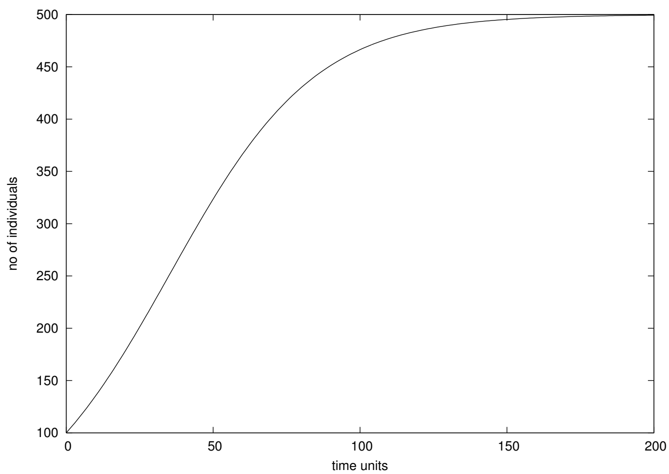



**Fig. 5.1** Logistic growth of a population ($\varrho = 4$, $M = 500$, $x_0 = 100$, $N = 200$).

If the equation stabilizes as $n \to \infty$, it means that $x_n = x_{n-1}$ in this limit. The equation then reduces to

$$x_n = x_n + \frac{\varrho}{100} x_n \left(1 - \frac{x_n}{M}\right) .$$

By inserting $x_n = M$ we see that this solution fulfills the equation. The same solution technique (i.e., setting $x_n = x_{n-1}$) can be used to check if $x_n$ in a difference equation approaches a limit or not.

Mathematical models like (5.12) are often easier to work with if we *scale* the variables, as briefly described in Chapter 4.7.2. Basically, this means that we divide each variable by a characteristic size of that variable such that the value of the new variable is typically 1. In the present case we can scale $x_n$ by $M$ and introduce a new variable,

$$y_n = \frac{x_n}{M} \, .$$

Similarly, $x_0$ is replaced by $y_0 = x_0/M$. Inserting $x_n = My_n$ in (5.12) and dividing by $M$ gives

$$y_n = y_{n-1} + qy_{n-1}\left(1 - y_{n-1}\right), \tag{5.13}$$

where $q = \varrho/100$ is introduced to save typing. Equation (5.13) is simpler than (5.12) in that the solution lies approximately between[5] $y_0$ and 1, and there are only two dimensionless input paramters to care about: $q$ and $y_0$. To solve (5.12) we need knowledge of three parameters: $x_0$, $\varrho$, and $M$.

### 5.1.6 Payback of a Loan

A loan $L$ is to be paid back over $N$ months. The payback in a month consists of the fraction $L/N$ plus the interest increase of the loan. Let the annual interest rate for the loan be $p$ percent. The monthly interest rate is then $\frac{p}{12}$. The value of the loan after month $n$ is $x_n$, and the change from $x_{n-1}$ can be modeled as

$$x_n = x_{n-1} + \frac{p}{12\cdot 100}x_{n-1} - \left(\frac{p}{12\cdot 100}x_{n-1} + \frac{L}{N}\right), \tag{5.14}$$

$$= x_{n-1} - \frac{L}{N}, \tag{5.15}$$

for $n = 1, \ldots, N$. The initial condition is $x_0 = L$. A major difference between (5.15) and (5.6) is that all terms in the latter are proportional to $x_n$ or $x_{n-1}$, while (5.15) also contains a constant term $(L/N)$. We say that (5.6) is homogeneous and linear, while (5.15) is inhomogeneous (because of the constant term) and linear. The mathematical solution of inhomogeneous equations are more difficult to find than the solution of homogeneous equations, but in a program there is no big difference: We just add the extra term $-L/N$ in the formula for the difference equation.

The solution of (5.15) is not particularly exciting[6]. What is more interesting, is what we pay each month, $y_n$. We can keep track of both $y_n$ and $x_n$ in a variant of the previous model:

$$y_n = \frac{p}{12\cdot 100}x_{n-1} + \frac{L}{N}, \tag{5.16}$$

$$x_n = x_{n-1} + \frac{p}{12\cdot 100}x_{n-1} - y_n \, . \tag{5.17}$$

[5] Values larger than 1 can occur, see Exercise 5.21.

[6] Use (5.15) repeatedly to derive the solution $x_n = L - nL/N$.

Equations (5.16)–(5.17) is a system of difference equations. In a computer code, we simply update $y_n$ first, and then we update $x_n$, inside a loop over $n$. Exercise 5.6 asks you to do this.

### 5.1.7 Taylor Series as a Difference Equation

Consider the following system of two difference equations

$$e_n = e_{n-1} + a_{n-1}, \tag{5.18}$$

$$a_n = \frac{x}{n} a_{n-1}, \tag{5.19}$$

with initial conditions $e_0 = 0$ and $a_0 = 1$. We can start to nest the solution:

$$\begin{aligned}
e_1 &= 0 + a_0 = 0 + 1 = 1,\\
a_1 &= x,\\
e_2 &= e_1 + a_1 = 1 + x,\\
a_2 &= \frac{x}{2}a_1 = \frac{x^2}{2},\\
e_3 &= e_2 + a_1 = 1 + x + \frac{x^2}{2},\\
e_4 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3\cdot 2},\\
e_5 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3\cdot 2} + \frac{x^4}{4\cdot 3\cdot 2}
\end{aligned}$$

The observant reader who has heard about Taylor series (see Chapter A.4) will recognize this as the Taylor series of $e^x$:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}\,. \tag{5.20}$$

How do we derive a system like (5.18)–(5.19) for computing the Taylor polynomial approximation to $e^x$? The starting point is the sum $\sum_{n=0}^{\infty} \frac{x^n}{n!}$. This sum is coded by adding new terms to an accumulation variable in a loop. The mathematical counterpart to this code is a difference equation

$$e_{n+1} = e_n + \frac{x^n}{n!}, \quad e_0 = 0,\ n = 0, 1, 2, \ldots\,. \tag{5.21}$$

or equivalently (just replace $n$ by $n-1$):

$$e_n = e_{n-1} + \frac{x^{n-1}}{n-1!}, \quad e_0 = 0,\ n = 1, 2, 3, \ldots\,. \tag{5.22}$$

Now comes the important observation: the term $x^n/n!$ contains many of the computations we already performed for the previous term $x^{n-1}/(n-1)!$ becase

$$x^n = \frac{x \cdot x \cdots x}{n(n-1)(n-2)\cdots 1}, \quad x^{n-1} = \frac{x \cdot x \cdots x}{(n-1)(n-2)(n-3)\cdots 1}\,.$$

Let $a_n = x^n/n!$. We see that we can go from $a_{n-1}$ to $a_n$ by multiplying $a_{n-1}$ by $x/n$:

$$\frac{x}{n}a_{n-1} = \frac{x}{n}\frac{x^{n-1}}{(n-1)!}\frac{x}{n} = \frac{x^n}{n!} = a_n, \tag{5.23}$$

which is nothing but (5.19). We also realize that $a_0 = 1$ is the initial condition for this difference equation. In other words, (5.18) sums the Taylor polynomial, and (5.19) updates each term in the sum.

The system (5.18)–(5.19) is very easy to implement in a program and constitutes an efficient way to compute (5.20). The function `exp_diffeq` does the work[7]:

```
def exp_diffeq(x, N):
    n = 1
    an_prev = 1.0  # a_0
    en_prev = 0.0  # e_0
    while n <= N:
        en = en_prev + an_prev
        an = x/n*an_prev
        en_prev = en
        an_prev = an
        n += 1
    return en
```

This function along with a direct evaluation of the Taylor series for $e^x$ and a comparison with the exact result for various $N$ values can be found in the file `exp_Taylor_series_diffeq.py`.

### 5.1.8 Making a Living from a Fortune

Suppose you want to live on a fortune $F$. You have invested the money in a safe way that gives an annual interest of $p$ percent. Every year you plan to consume an amount $c_n$, where $n$ counts years. The development of your fortune $x_n$ from one year to the other can then be modeled by

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F\,. \tag{5.24}$$

A simple example is to keep $c$ constant, say $q$ percent of the interest the first year:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - \frac{pq}{10^4}F, \quad x_0 = F\,. \tag{5.25}$$

[7] Observe that we do not store the sequences in arrays, but make use of the fact that only the most recent sequence element is needed to calculate a new element.

A more realistic model is to assume some inflation of $I$ percent per year. You will then like to increase $c_n$ by the inflation. We can extend the model in two ways. The simplest and clearest way, in the author's opinion, is to track the evolution of two sequences $x_n$ and $c_n$:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F,\ c_0 = \frac{pq}{10^4}F, \tag{5.26}$$

$$\tag{5.27}$$

$$c_n = c_{n-1} + \frac{I}{100}c_{n-1}\,. \tag{5.28}$$

This is a system of two difference equations with two unknowns. The solution method is, nevertheless, not much more complicated than the method for a difference equation in one unknown, since we can first compute $x_n$ from (5.27) and then update the $c_n$ value from (5.28). You are encouraged to write the program (see Exercise 5.7).

Another way of making a difference equation for the case with inflation, is to use an explicit formula for $c_{n-1}$, i.e., solve (5.27) and end up with a formula like (5.4). Then we can insert the explicit formula

$$c_{n-1} = \left(1 + \frac{I}{100}\right)^{n-1} \frac{pq}{10^4}F$$

in (5.24), resulting in only one difference equation to solve.

### 5.1.9 Newton's Method

The difference equation

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad x_0 \text{ given}, \tag{5.29}$$

generates a sequence $x_n$ where, if the sequence converges (i.e., if $x_n - x_{n-1} \to 0$), $x_n$ approaches a root of $f(x)$. That is, $x_n \to x$, where $x$ solves the equation $f(x) = 0$. Equation (5.29) is the famous Newton's method for solving nonlinear algebraic equations $f(x) = 0$. When $f(x)$ is not linear, i.e., $f(x)$ is not on the form $ax + b$ with constant $a$ and $b$, (5.29) becomes a *nonlinear difference equation*. This complicates analytical treatment of difference equations, but poses no extra difficulties for numerical solution.

We can quickly sketch the derivation of (5.29). Suppose we want to solve the equation

$$f(x) = 0$$

and that we already have an approximate solution $x_{n-1}$. If $f(x)$ were linear, $f(x) = ax+b$, it would be very easy to solve $f(x) = 0$: $x = -b/a$. The idea is therefore to approximate $f(x)$ in the vicinity of $x = x_{n-1}$ by a linear function, i.e., a straight line $f(x) \approx \tilde{f}(x) = ax + b$. This

line should have the same slope as $f(x)$, i.e., $a = f'(x_{n-1})$, and both the line and $f$ should have the same value at $x = x_{n-1}$. From this condition one can find $b = f(x_{n-1}) - x_{n-1}f'(x_{n-1})$. The approximate function (line) is then

$$\tilde{f}(x)f(x_{n-1}) + f'(x_{n-1})(x - x_{n-1})\,. \tag{5.30}$$

This expression is just the two first terms of a Taylor series approximation to $f(x)$ at $x = x_{n-1}$. It is now easy to solve $\tilde{f}(x) = 0$ with respect to $x$, and we get

$$x = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}\,. \tag{5.31}$$

Since $\tilde{f}$ is only an approximation to $f$, $x$ in (5.31) is only an approximation to a root of $f(x) = 0$. Hopefully, the approximation is better than $x_{n-1}$ so we set $x_n = x$ as the next term in a sequence that we hope converges to the correct root. However, convergence depends highly on the shape of $f(x)$, and there is no guarantee that the method will work.

The previous programs for solving difference equations have typically calculated a sequence $x_n$ up to $n = N$, where $N$ is given. When using (5.29) to find roots of nonlinear equations, we do not know a suitable $N$ in advance that leads to an $x_n$ where $f(x_n)$ is sufficiently close to zero. We therefore have to keep on increasing $n$ until $f(x_n) < \epsilon$ for some small $\epsilon$. Of course, the sequence diverges, we will keep on forever, so there must be some maximum allowable limit on $n$, which we may take as $N$.

It can be convenient to have the solution of (5.29) as a function for easy reuse. Here is a first rough implementation:

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100):
    n = 0
    while abs(f(x)) > epsilon and n <= N:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, n, f(x)
```

This function might well work, but `f(x)/dfdx(x)` can imply integer division, so we should ensure that the numerator or denumerator is of `float` type. There are also two function evaluations of `f(x)` in every pass in the loop (one in the loop body and one in the `while` condition). We can get away with only one evaluation if we store the `f(x)` in a local variable. In the small examples with $f(x)$ in the present course, twice as many function evaluations of $f$ as necessary does not matter, but the same `Newton` function can in fact be used for much more complicated functions, and in those cases twice as much work can be noticable. As a programmer, you should therefore learn to optimize the code by removing unnecessary computations.

Another, more serious, problem is the possibility dividing by zero. Almost as serious, is dividing by a very small number that creates a large value, which might cause Newton's method to diverge. Therefore, we should test for small values of $f'(x)$ and write a warning or raise an exception.

Another improvement is to add a boolean argument `store` to indicate whether we want the $(x, f(x))$ values during the iterations to be stored in a list or not. These intermediate values can be handy if we want to print out or plot the convergence behavior of Newton's method.

An improved `Newton` function can now be coded as

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100, store=False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= N:
        dfdx_value = float(dfdx(x))
        if abs(dfdx_value) < 1E-14:
            raise ValueError("Newton: f'(%g)=%g" % (x, dfdx_value))

        x = x - f_value/dfdx_value

        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```

Note that to use the `Newton` function, we need to calculate the derivative $f'(x)$ and implement it as a Python function and provide it as the `dfdx` argument. Also note that what we return depends on whether we store $(x, f(x))$ information during the iterations or not.

It is quite common to test if `dfdx(x)` is zero in an implementation of Newton's method, but this is not strictly necessary in Python since an exception `ZeroDivisionError` is always raised when dividing by zero.

We can apply the `Newton` function to solve the equation[8] $e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0$:

```
from math import sin, cos, exp, pi
import sys
from Newton import Newton

def g(x):
    return exp(-0.1*x**2)*sin(pi/2*x)

def dg(x):
    return -2*0.1*x*exp(-0.1*x**2)*sin(pi/2*x) + \
           pi/2*exp(-0.1*x**2)*cos(pi/2*x)

x0 = float(sys.argv[1])
x, info = Newton(g, x0, dg, store=True)
```

[8] Fortunately you realize that the exponential function can never be zero, so the solutions of the equation must be the zeros of the sine function, i.e., $\frac{\pi}{2}x = i\pi$ for all integers $i = \dots, -2, 1, 0, 1, 2, \dots$. This gives $x = 2i$ as the solutions.

```
print 'root:', x
for i in range(len(info)):
    print 'Iteration %3d: f(%g)=%g' % \
          (i, info[i][0], info[i][1])
```

The Newton function and this program can be found in the file Newton.py. Running this program with an initial $x$ value of 1.7 results in the output

```
root: 1.999999999768449
Iteration  0: f(1.7)=0.340044
Iteration  1: f(1.99215)=0.00828786
Iteration  2: f(1.99998)=2.53347e-05
Iteration  3: f(2)=2.43808e-10
```

The convergence is fast towards the solution $x = 2$. The error is of the order $10^{-10}$ even though we stop the iterations when $f(x) \leq 10^{-7}$.

Trying a start value of 3 we would expect the method to find either nearby solution $x = 2$ or $x = 4$, but now we get

```
root: 42.49723316011362
Iteration  0: f(3)=-0.40657
Iteration  1: f(4.66667)=0.0981146
Iteration  2: f(42.4972)=-2.59037e-79
```

We have definitely solved $f(x) = 0$ in the sense that $|f(x)| \leq \epsilon$, where $\epsilon$ is a small value (here $\epsilon \sim 10^{-79}$). However, the solution $x \approx 42.5$ is *not* close to the solution ($x = 42$ and $x = 44$ are the solutions closest to the computed $x$). Can you use your knowledge of how the Newton method works and figure out why we get such strange behavior?

The demo program Newton_movie.py can be used to investigate the strange behavior. This program takes five command-line arguments: a formula for $f(x)$, a formula for $f'(x)$ (or the word numeric, which indicates a numerical approximation of $f'(x)$), a guess at the root, and the minimum and maximum $x$ values in the plots. We try the following case with the program:

Terminal

```
Newton_movie.py 'exp(-0.1*x**2)*sin(pi/2*x)' numeric 3 -3 43
```

As seen, we start with $x = 3$ as the initial guess. In the first step of the method, we compute a new value of the root, now $x = 4.66667$. As we see in Figure 5.2, this root is near an extreme point of $f(x)$ so that the derivative is small, and the resulting straight line approximation to $f(x)$ at this root becomes quite flat. The result is a new guess at the root: $x42.5$. This root is far away from the last root, but the second problem is that $f(x)$ is quickly damped as we move to increasing $x$ values, and at $x = 42.5$ $f$ is small enough to fulfill the convergence criterion. Any guess at the root out in this region would satisfy that criterion.

You can run the Newton_movie.py program with other values of the initial root and observe that the method usually finds the nearest roots.

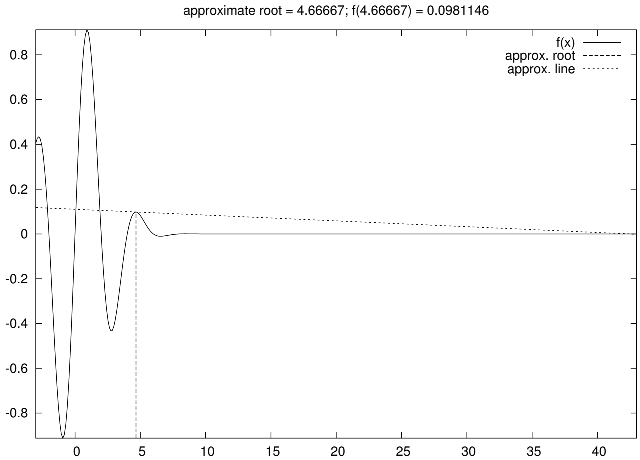



**Fig. 5.2** Failure of Newton's method to solve $e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0$. The plot corresponds to the second root found (starting with $x = 3$).

### 5.1.10 The Inverse of a Function

Given a function $f(x)$, the inverse function of $f$, say we call it $g(x)$, has the property that if we apply $g$ to the value $f(x)$, we get $x$ back:

$$g(f(x)) = x\,.$$

Similarly, if we apply $f$ to the value $g(x)$, we get $x$:

$$f(g(x)) = x\,. \tag{5.32}$$

By hand, you substitute $g(x)$ by (say) $y$ in (5.32) and solve (5.32) with respect to $y$ to find some $x$ expression for the inverse function. For example, given $f(x) = x^2 - 1$, we must solve $y^2 - 1 = x$ with respect to $y$. To ensure a unique solution for $y$, the $x$ values have to be limited to an interval where $f(x)$ is monotone, say $x \in [0, 1]$ in the present example. Solving for $y$ gives $y = \sqrt{1+x}$, therefore and $g(x) = \sqrt{1+x}$. It is easy to check that $f(g(x)) = (\sqrt{1+x})^2 - 1 = x$.

Numerically, we can use the "definition" (5.32) of the inverse function $g$ at one point at a time. Suppose we have a sequence of points $x_0 < x_1 < \cdots < x_N$ along the $x$ axis such that $f$ is monotone in $[x_0, x_N]$: $f(x_0) > f(x_1) > \cdots > f(x_N)$ or $f(x_0) < f(x_1) < \cdots < f(x_N)$. For each point $x_i$, we have

$$f(g(x_i)) = x_i\,.$$

The value $g(x_i)$ is unknown, so let us call it $\gamma$. The equation

$$f(\gamma) = x_i \tag{5.33}$$

can be solved be respect $\gamma$. However, (5.33) is in general nonlinear if $f$ is a nonlinear function of $x$. We must then use, e.g., Newton's method

to solve (5.33). Newton's method works for an equation phrased as "$f(x) = 0$", which in our case is $f(\gamma) - x_i = 0$, i.e., we seek the roots of the function $F(\gamma) \equiv f(\gamma) - x_i$. Also the derivative $F'(\gamma)$ is needed in Newton's method. For simplicity we may use an approximate finite difference:

$$\frac{dF}{d\gamma} \approx \frac{F(\gamma + h) - F(\gamma - h)}{2h} .$$

As start value $\gamma_0$, we can use the previously computed $g$ value: $g_{i-1}$. We introduce the short notation $\gamma = \text{Newton}(F, \gamma_0)$ to indicate the solution of $F(\gamma) = 0$ with initial guess $\gamma_0$.

The computation of all the $g_0, \ldots, g_N$ values can now be expressed by

$$g_i = \text{Newton}(F, g_{i-1}), \quad i = 1, \ldots, N, \tag{5.34}$$

and for the first point we may use $x_0$ as start value (for instance):

$$g_0 = \text{Newton}(F, x_0) . \tag{5.35}$$

Equations (5.34)–(5.35) constitute a difference equation for $g_i$, since given $g_{i-1}$, we can compute the next element of the sequence by (5.34). Because (5.34) is a nonlinear equation in the new value $g_i$, and (5.34) is therefore an example of a *nonlinear difference equation*.

The following program computes the inverse function $g(x)$ of $f(x)$ at some discrete points $x_0, \ldots, x_N$. Our sample function is $f(x) = x^2 - 1$:

```
from Newton import Newton
from scitools.std import *

def f(x):
    return x**2 - 1

def F(gamma):
    return f(gamma) - xi

def dFdx(gamma):
    return (F(gamma+h) - F(gamma-h))/(2*h)

h = 1E-6
x = linspace(0.01, 3, 21)
g = zeros(len(x))

for i in range(len(x)):
    xi = x[i]

    # compute start value (use last g[i-1] if possible):
    if i == 0:
        gamma0 = x[0]
    else:
        gamma0 = g[i-1]

    gamma, n, F_value = Newton(F, gamma0, dFdx)
    g[i] = gamma

plot(x, f(x), 'r-', x, g, 'b-',
     title='f1', legend=('original', 'inverse'))
```

Note that with $f(x) = x^2 - 1$, $f'(0) = 0$, so Newton's method divides by zero and breaks down unless with let $x_0 > 0$, so here we set $x_0 = 0.01$. The `f` function can easily be edited to let the program compute the inverse of another function. The `F` function can remain the same since it applies a general finite difference to approximate the derivative of the `f(x)` function. The complete program is found in the file `inverse_function.py`. A better implementation is suggested in Exercise 7.20.

## 5.2 Programming with Sound

Sound on a computer is nothing but a sequence of numbers. As an example, consider the famous A tone at 440 Hz. Physically, this is an oscillation of a tunefork, loudspeaker, string or another mechanical medium that makes the surrounding air also oscillate and transport the sound as a compression wave. This wave may hit our ears and through complicated physiological processes be transformed to an electrical signal that the brain can recognize as sound. Mathematically, the oscillations are described by a sine function of time:

$$s(t) = A\sin(2\pi f t)\,, \tag{5.36}$$

where $A$ is the amplitude or strength of the sound and $f$ is the frequencey (440 Hz for the A in our example). In a computer, $s(t)$ is represented at discrete points of time. CD quality means 44100 samples per second. Other sample rates are also possible, so we introduce $r$ as the sample rate. An $f$ Hz tone lasting for $m$ seconds with sample rate $r$ can then be computed as the sequence

$$s_n = A\sin\left(2\pi f\frac{n}{r}\right), \quad n = 0, 1, \ldots, m\cdot r\,. \tag{5.37}$$

With Numerical Python this computation is straightforward and very efficient. Introducing some more explanatory variable names than $r$, $A$, and $m$, we can write a function for generating a note:

```
import numpy
def note(frequency, length, amplitude=1, sample_rate=44100):
    time_points = numpy.linspace(0, length, length*sample_rate)
    data = numpy.sin(2*numpy.pi*frequency*time_points)
    data = amplitude*data
    return data
```

### 5.2.1 Writing Sound to File

The `note` function above generates an array of `float` data representing a note. The sound card in the computer cannot play these data, because

the card assumes that the information about the oscillations appears as a sequence of two-byte integers. With an array's `astype` method we can easily convert our data to two-byte integers instead of `floats`:

```
data = data.astype(numpy.int16)
```

That is, the name of the two-byte integer data type in `numpy` is `int16` (two bytes are 16 bits). The maximum value of a two-byte integer is $2^{15} - 1$, so this is also the maximum amplitude. Assuming that `amplitude` in the `note` function is a relative measure of intensity, such that the value lies between 0 and 1, we must adjust this amplitude to the scale of two-byte integers:

```
max_amplitude = 2**15 - 1
data = max_amplitude*data
```

The `data` array of `int16` numbers can be written to a file and played as an ordinary file in CD quality. Such a file is known as a wave file or simply a WAV file since the extension is `.wav`. Python has a module `wave` for creating such files. Given an array of sound, `data`, we have in SciTools a module `sound` with a function `write` for writing the data to a WAV file (using functionality from the `wave` module):

```
import scitools.sound
scitools.sound.write(data, 'Atone.wav')
```

You can now use your favorite music player to play the `Atone.wav` file, or you can play it from within a Python program using

```
scitools.sound.play('Atone.wav')
```

The `write` function can take more arguments and write, e.g., a stereo file with two channels, but we do not dive into these details here.

### 5.2.2 Reading Sound from File

Given a sound signal in a WAV file, we can easily read this signal into an array and mathematically manipulate the data in the array to change the flavor of the sound, e.g., add echo, treble, or bass. The recipe for reading a WAV file with name `filename` is

```
data = scitools.sound.read(filename)
```

The `data` array has elements of type `int16`. Often we want to compute with this array, and then we need elements of `float` type, obtained by the conversion

```
data = data.astype(float)
```

The `write` function automatically transforms the element type back to `int16` if we have not done this explicitly.

One operation that we can easily do is adding an echo. Mathematically this means that we add a damped delayed sound, where the original sound has weight $\beta$ and the delayed part has weight $1-\beta$, such that the overall amplitude is not altered. Let $d$ be the delay in seconds. With a sampling rate $r$ the number of indices in the delay becomes $dr$, which we denote by $b$. Given an original sound sequence $s_n$, the sound with echo is the sequence

$$e_n = \beta s_n + (1-\beta)s_{n-b}\,. \tag{5.38}$$

We cannot start $n$ at 0 since $e_0 = s_{0-b} = s_{-b}$ which is a value outside the sound data. Therefore we define $e_n = s_n$ for $n = 0, 1, \ldots, b$, and add the echo thereafter. A simple loop can do this (again we use descriptive variable names instead of the mathematical symbols introduced):

```
def add_echo(data, beta=0.8, delay=0.002, sample_rate=44100):
    newdata = data.copy()
    shift = int(delay*sample_rate)  # b (math symbol)
    for i in range(shift, len(data)):
        newdata[i] = beta*data[i] + (1-beta)*data[i-shift]
    return newdata
```

The problem with this function is that it runs slowly, especially when we have sound clips lasting several seconds (recall that for CD quality we need 44100 numbers per second). It is therefore necessary to vectorize the implementation of the difference equation for adding echo. The update is then based on adding slices:

```
newdata[shift:] = beta*data[shift:] + \
                  (1-beta)*data[:len(data)-shift]
```

### 5.2.3 Playing Many Notes

How do we generate a melody mathematically in a computer program? With the `note` function we can generate a note with a certain amplitude, frequence, and duration. The note is represented as an array. Putting sound arrays for different notes after each other will make up a melody. If we have several sound arrays `data1`, `data2`, `data3`, ..., we can make a new array consisting of the elements in the first array followed by the elements of the next array followed by the elements in the next array and so forth:

```
data = numpy.concatenate((data1, data2, data3, ...))
```

Here is an example of creating a little melody (start of "Nothing Else Matters" by Metallica) using constant (max) amplitude of all the notes:

```
E1 = note(164.81, .5)
G = note(392, .5)
B = note(493.88, .5)
E2 = note(659.26, .5)
intro = numpy.concatenate((E1, G, B, E2, B, G))
high1_long = note(987.77, 1)
high1_short = note(987.77, .5)
high2 = note(1046.50, .5)
high3 = note(880, .5)
high4_long = note(659.26, 1)
high4_medium = note(659.26, .5)
high4_short = note(659.26, .25)
high5 = note(739.99, .25)
pause_long =  note(0, .5)
pause_short = note(0, .25)
song = numpy.concatenate(
  (intro, intro, high1_long, pause_long, high1_long,
   pause_long, pause_long,
   high1_short, high2, high1_short, high3, high1_short,
   high3, high4_short, pause_short, high4_long, pause_short,
   high4_medium, high5, high4_short))
scitools.sound.play(song)
scitools.sound.write(song, 'tmp.wav')
```

We could send `song` to the `add_echo` function to get some echo, and we could also vary the amplitudes to get more dynamics into the song. You can find the generation of notes above as the function `Nothing_Else_Matters(echo=False)` in the `scitools.sound` module.

## 5.3 Summary

### 5.3.1 Chapter Topics

*Sequences.* In general, a finite sequence can be written as

$$(x_n)_{n=0}^{N}, \quad x_n = f(n),$$

where $f(n)$ is some expression involving $n$ and possibly other parameters. The coding of such a sequence takes the form

```
x = zeros(N+1)
for n in range(N+1):
    x[n] = f(n)
```

Here, we store the whole sequence in an array, which is convenient if we want to plot the evolution of the sequence (i.e., $x_n$ versus $n$). Occasionally, this can require too much memory. Especially when checking for convergence of $x_n$ toward some limit as $N \to \infty$, $N$ may be large

if the sequence converges slowly. The array can be skipped if we print the sequence elements as they are computed:

```
for n in range(N+1):
    print f(n)
```

*Difference Equations.* Equations involving more than one element of a sequence are called difference equations. We have typically looked at difference equations relating $x_n$ to the previous element $x_{n-1}$:

$$x_n = f(x_{n-1}),$$

where $f$ is some specified function. Any difference equation must have an initial condition prescribing $x_0$, say $x_0 = X$. The solution of a difference equation is a sequence. Very often, $n$ corresponds to a time parameter in applications.

We have also looked at systems of difference equations of the form

$$\begin{aligned} x_n &= f(x_{n-1}, y_{n-1}), \quad x_0 = X \\ y_n &= g(y_{n-1}, x_{n-1}, x_n), \quad y_0 = Y \end{aligned}$$

where $f$ and $g$ denote formulas involving already computed quantities. Note that $x_n$ does not depend on $y_n$, which means that we can first compute $x_n$ and then $y_n$:

```
index_set = range(N+1)
x = zeros(len(index_set))
y = zeros(len(index_set))
x[0] = X
y[0] = Y
for n in index_set[1:]:
    x[n] = f(x[n-1], y[n-1])
    y[n] = g(y[n-1], x[n], x[n-1])
```

*Sound.* Sound on a computer is a sequence of 2-byte integers. These can be stored in an array. Creating the sound of a tone consist of sampling a sine function and storing the values in an array (and converting to 2-byte integers). If we want to manipulate a given sound, say add some echo, we convert the array elements to ordinary floating-point numbers and peform mathematical operations on the array elements.

### 5.3.2 Summarizing Example: Music of a Sequence

*Problem.* The purpose of this summarizing example is to listen to the sound generated by two mathematical sequences. The first one is given by an explicit formula, constructed to oscillate around 0 with decreasing amplitude:

$$x_n = e^{-4n/N}\sin(8\pi n/N)\,. \tag{5.39}$$

The other sequence is generated by the difference equation (5.13) for logistic growth, repeated here for convenience:

$$x_n = x_{n-1} + qx_{n-1}\left(1 - x_{n-1}\right), \quad x = x_0\,. \tag{5.40}$$

We let $x_0 = 0.01$ and $q = 2$. This leads to fast initial growth toward the limit 1, and then oscillations around this limit (this problem is studied in Exercise 5.21).

The absolute value of the sequence elements $x_n$ are of size between 0 and 1, approximately. We want to transform these sequence elements to tones, using the techniques of Chapter 5.2. First we convert $x_n$ to a frequency the human ear can hear. The transformation

$$y_n = 440 + 200x_n \tag{5.41}$$

will make a standard A reference tone out of $x_n = 0$, and for the maximum value of $x_n$ around 1 we get a tone of 640 Hz. Elements of the sequence generated by (5.39) lie between -1 and 1, so the corresponding frequences lie between 240 Hz and 640 Hz. The task now is to make a program that can generate and play the sounds.

*Solution.* Tones can be generated by the `note` function from the `scitools.sound` module. We collect all tones corresponding to all the $y_n$ frequencies in a list `tones`. Letting `N` denote the number of sequence elements, the relevant code segment reads

```
from scitools.sound import *
freqs = 440 + x*200
tones = []
duration = 30.0/N     # 30 sec sound in total
for n in range(N+1):
    tones.append(max_amplitude*note(freqs[n], duration, 1))
data = concatenate(tones)
write(data, filename)
data = read(filename)
play(filename)
```

It is illustrating to plot the sequences too,

```
plot(range(N+1), freqs, 'ro')
```

To generate the sequences (5.39) and (5.40), we make two functions, `oscillations` and `logistic`, respectively. These functions take the number of sequence elements (`N`) as input and return the sequence stored in an array.

In another function `make_sound` we compute the sequence, transform the elements to frequencies, generate tones, write the tones to file, and play the sound file.

As always, we collect the functions in a module and include a test block where we can read the choice of sequence and the sequence length from the command line. The complete module file look as follows:

```
from scitools.sound import *
from scitools.std import *

def oscillations(N):
    x = zeros(N+1)
    for n in range(N+1):
        x[n] = exp(-4*n/float(N))*sin(8*pi*n/float(N))
    return x

def logistic(N):
    x = zeros(N+1)
    x[0] = 0.01
    q = 2
    for n in range(1, N+1):
        x[n] = x[n-1] + q*x[n-1]*(1 - x[n-1])
    return x

def make_sound(N, seqtype):
    filename = 'tmp.wav'
    x = eval(seqtype)(N)
    # convert x values to frequences around 440:
    freqs = 440 + x*200
    plot(range(N+1), freqs, 'ro')
    # generate tones:
    tones = []
    duration = 30.0/N     # 30 sec sound in total
    for n in range(N+1):
        tones.append(max_amplitude*note(freqs[n], duration, 1))
    data = concatenate(tones)
    write(data, filename)
    data = read(filename)
    play(filename)

if __name__ == '__main__':
    try:
        seqtype = sys.argv[1]
        N = int(sys.argv[2])
    except IndexError:
        print 'Usage: %s oscillations|logistic N' % sys.argv[0]
        sys.exit(1)
    make_sound(N, seqtype)
```

This code should be quite easy to read at the present stage in the book. However, there is one statement that deserves a comment:

```
    x = eval(seqtype)(N)
```

The `seqtype` argument reflects the type of sequence and is a string that the user provides on the command line. The values of the string equal the function names `oscillations` and `logistic`. With `eval(seqtype)` we turn the string into a function name. For example, if `seqtype` is `'logistic'`, performing an `eval(seqtype)(N)` is the same as if we had written `logistic(N)`. This technique allows the user of the program to choose a function call inside the code. Without `eval` we would need to explicitly test on values:

```
if seqtype == 'logistic':
    x = logistic(N)
elif seqtype == 'oscillations':
    x = oscillations(N)
```

This is not much extra code to write in the present example, but if we have a large number of functions generating sequences, we can save a lot of boring if-else code by using the `eval` construction.

The next step, as a reader who have understood the problem and the implementation above, is to run the program for two cases: the `oscillations` sequence with $N = 40$ and the `logistic` sequence with $N = 100$. By altering the $q$ parameter to lower values, you get other sounds, typically quite boring sounds for non-oscillating logistic growth ($q < 1$). You can also experiment with other transformations of the form (5.41), e.g., increasing the frequency variation from 200 to 400.

## 5.4 Exercises

**Exercise 5.1.** *Determine the limit of a sequence.*

Given the sequence

$$a_n = \frac{7 + 1/n}{3 - 1/n^2},$$

make a program that computes and prints out $a_n$ for $n = 1, 2, \ldots, N$. Read $N$ from the command line. Does $a_n$ approach a finite limit when $n \to \infty$? Name of program file: `sequence_limit1.py`. ◇

**Exercise 5.2.** *Determine the limit of a sequence.*

Solve Exercise 5.1 when the sequence of interest is given by

$$D_n = \frac{\sin(2^{-n})}{2^{-n}}\,.$$

Name of program file: `sequence_limit2.py`. ◇

**Exercise 5.3.** *Experience convergence problems.*

Given the sequence

$$D_n = \frac{f(x+h) - f(x)}{h}, \quad h = 2^{-n} \tag{5.42}$$

make a function `D(f, x, N)` that takes a function $f(x)$, a value $x$, and the number $N$ of terms in the sequence as arguments, and returns an array with the $D_n$ values for $n = 0, 1, \ldots, N-1$. Make a call to the `D` function with $f(x) = \sin x$, $x = 0$, and $N = 80$. Plot the evolution of the computed $D_n$ values, using small circles for the data points.

Make another call to `D` where $x = \pi$ and plot this sequence in a separate figure. What would be your expected limit? Why do the com-

putations go wrong for large $N$? (Hint: Print out the numerator and denominator in $D_n$.) Name of program file: `sequence_limits3.py`. ◇

**Exercise 5.4.** *Convergence of sequences with $\pi$ as limit.*
The following sequences all converge to $\pi$:

$$(a_n)_{n=1}^{\infty}, \quad a_n = 4\sum_{k=1}^{n}\frac{(-1)^{k+1}}{2k-1},$$

$$(b_n)_{n=1}^{\infty}, \quad b_n = \left(6\sum_{k=1}^{n}k^{-2}\right)^{1/2},$$

$$(c_n)_{n=1}^{\infty}, \quad c_n = \left(90\sum_{k=1}^{n}k^{-4}\right)^{1/4},$$

$$(d_n)_{n=1}^{\infty}, \quad d_n = \frac{6}{\sqrt{3}}\sum_{k=0}^{n}\frac{(-1)^k}{3^k(2k+1)},$$

$$(e_n)_{n=1}^{\infty}, \quad e_n = 16\sum_{k=0}^{n}\frac{(-1)^k}{5^{2k+1}(2k+1)} - 4\sum_{k=0}^{n}\frac{(-1)^k}{239^{2k+1}(2k+1)}.$$

Make a function for each sequence that returns an array with the elements in the sequence. Plot all the sequences, and find the one that converges fastest toward the limit $\pi$. Name of program file: `pi.py`. ◇

**Exercise 5.5.** *Reduce memory usage of difference equations.*
Consider the program `growth_years.py` from Chapter 5.1.1. Since $x_n$ depends on $x_{n-1}$ only, we do not need to store all the $N+1$ $x_n$ values. We actually only need to store $x_n$ and its previous value $x_{n-1}$. Modify the program to use two variables for $x_n$ and not an array. Also avoid the `index_set` list and use an integer counter for $n$ and a `while` instead. (Of course, without the arrays it is not possible to plot the development of $x_n$, so you have to remove the `plot` call.) Name of program file: `growth_years_efficient.py`. ◇

**Exercise 5.6.** *Development of a loan over $N$ months.*
Solve (5.16)–(5.17) for $n = 1, 2, \ldots, N$ in a Python function. Name of program file: `loan.py`. ◇

**Exercise 5.7.** *Solve a system of difference equations.*
Solve (5.27)–(5.28) by generating the $x_n$ and $c_n$ sequences in a Python function. Let the function return the computed sequences as arrays. Plot the $x_n$ sequence. Name of program file: `fortune_and_inflation1.py`. ◇

**Exercise 5.8.** *Extend the model (5.27)–(5.28).*
In the model (5.27)–(5.28) the new fortune is the old one, plus the interest, minus the consumption. During year $n$, $x_n$ is normally also

reduced with $t$ percent tax on the earnings $x_{n-1} - x_{n-2}$ in year $n-1$. Extend the model with an appropriate tax term, modify the program from Exercise 5.7, and plot $x_n$ with tax ($t = 28$) and without tax ($t = 0$). Name of program file: `fortune_and_inflation2.py`. ◇

**Exercise 5.9.** *Experiment with the program from Exer. 5.8.*

Suppose you expect to live for $N$ years and can accept that the fortune $x_n$ vanishes after $N$ years. Experiment with the program from Exercise 5.8 for how large the initial $c_0$ can be in this case. Choose some appropriate values for $p$, $q$, $I$, and $t$. Name of program file: `fortune_and_inflation3.py`. ◇

**Exercise 5.10.** *Change index in a difference equation.*

A mathematically equivalent equation to (5.5) is

$$x_{i+1} = x_i + \frac{p}{100}x_i, \tag{5.43}$$

since the name of the index can be chosen arbitrarily. Suppose someone has made the following program for solving (5.43) by a slight editing of the program `growth1.py`:

```
from scitools.std import *
x0 = 100                # initial amount
p = 5                   # interest rate
N = 4                   # number of years
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
for i in index_set[1:]:
    x[i+1] = x[i] + (p/100.0)*x[i]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

This program does not work. Make a correct version, but keep the difference equations in its present form with the indices `i+1` and `i`. Name of program file: `growth1_index_ip1.py`. ◇

**Exercise 5.11.** *Construct time points from dates.*

A certain quantity $p$ (which may be an interest rate) is piecewise constant and undergoes changes at some specific dates, e.g.,

$$p \text{ changes to } \begin{cases} 4.5 & \text{on Jan 4, 2009} \\ 4.75 & \text{on March 21, 2009} \\ 6.0 & \text{on April 1, 2009} \\ 5.0 & \text{on June 30, 2009} \\ 4.5 & \text{on Nov 1, 2009} \\ 2.0 & \text{on April 1, 2010} \end{cases} \tag{5.44}$$

Given a start date $d_1$ and an end date $d_2$, fill an array `p` with the right $p$ values, where the array index counts days. Use the `datetime` module to

compute the number of days between dates and store the information about changes in $p$ in a list of 2-tuples, where each 2-tuple holds the date of change and the corresponding value of $p$. For the changes in $p$ given above, the list becomes

```
change_in_p = [(datetime.date(2009, 1, 4), 4.5),
               (datetime.date(2009, 3, 21), 4.75),
               (datetime.date(2009, 4, 1), 6.0),
               (datetime.date(2009, 6, 30), 5.0),
               (datetime.date(2009, 11, 1), 4.5),
               (datetime.date(2010, 4, 1), 2.0)]
```

Name of program file: `dates2days.py`. ◇

**Exercise 5.12.** *Solve nonlinear equations by Newton's method.*
Import the `Newton` function from the `Newton.py` file from Chapter 5.1.9 to solve the following nonlinear algebraic equations:

$$\sin x = 0, \tag{5.45}$$
$$x = \sin x, \tag{5.46}$$
$$x^5 = \sin x, \tag{5.47}$$
$$x^4 \sin x = 0, \tag{5.48}$$
$$x^4 = 0, \tag{5.49}$$
$$x^{10} = 0, \tag{5.50}$$
$$\tanh x = x^{10}\,. \tag{5.51}$$

Read the starting point $x_0$ and the equation to be solved from the command line (use `StringFunction` from Chapter 3.1.4 to convert the formula for $f(x)$ to a Python function). Print out the evolution of the roots (based on the `info` list). You will need to carefully plot the $f(x)$ function to understand how Newton's method will behave in each case for different starting values $x_0$. Find an $x_0$ value for each equation so that Newton's method will converge toward the root $x = 0$. Name of program file: `Newton_examples.py`. ◇

**Exercise 5.13.** *Visualize the convergence of Newton's method.*
Let $x_0, x_1, \ldots, x_N$ be the sequence of roots generated by Newton's method applied to a nonlinear algebraic equation $f(x) = 0$ (cf. Chapter 5.1.9). In this exercise, the purpose is to plot the sequences $(x_n)_{n=0}^N$ and $(f(x_n))_{n=0}^N$. Make a general function `Newton_plot(f, x, dfdx, epsilon=1E-7)` for this purpose. The first two arguments, `f` and `dfdx`, are Python functions representing the $f(x)$ function in the equation and its derivative $f'(x)$, respectively. Newton's method is run until $|f(x_N)| \leq \epsilon$, and the $\epsilon$ value is the third argument (`epsilon`). The `Newton_plot` function should make one plot of $(x_n)_{n=0}^N$ and $(f(x_n))_{n=0}^N$ on the screen and one save to a PNG file. (Hint: You can save quite some coding by calling the improved `Newton`

function from Chapter 5.1.9, which is available in the `Newton` module in `src/diffeq/Newton.py`.)

Demonstrate the function on the equation $x^6 \sin \pi x = 0$, with $\epsilon = 10^{-13}$. Try different starting values for Newton's method: $x_0 = -2.6, -1.2, 1.5, 1.7, 0.6$. Compare the results with the exact solutions $x = \ldots, -2 - 1, 0, 1, 2, \ldots$. Name of program file: `Newton2.py`. ◇

**Exercise 5.14.** *Implement the Secant method.*

Newton's method (5.29) for solving $f(x) = 0$ requires the derivative of the function $f(x)$. Sometimes this is difficult or inconvenient. The derivative can be approximated using the last two approximations to the root, $x_{n-2}$ and $x_{n-1}$:

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}} .$$

Using this approximation in (5.29) leads to the Secant method:

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}, \quad x_0, x_1 \text{ given} . \tag{5.52}$$

Here $n = 2, 3, \ldots$. Make a program that applies the Secant method to solve $x^5 = \sin x$. Name of program file: `Secant.py`. ◇

**Exercise 5.15.** *Test different methods for root finding.*

Make a program for solving $f(x) = 0$ by Newton's method (Chapter 5.1.9), the Bisection method (Chapter 3.6.2), and the Secant method (Exercise 5.14). For each method, the sequence of root approximations should be written out (nicely formatted) on the screen. Read $f(x)$, $a$, $b$, $x_0$, and $x_1$ from the command line. Newton's method starts with $x_0$, the Bisection method starts with the interval $[a, b]$, whereas the Secant method starts with $x_0$ and $x_1$.

Run the program for each of the equations listed in Exercise 5.12. You should first plot the $f(x)$ functions as suggested in that exercise so you know how to choose $x_0$, $x_1$, $a$, and $b$ in each case. Name of program file: `root_finder_examples.py`. ◇

**Exercise 5.16.** *Difference equations for computing* $\sin x$*.*

The purpose of this exercise is to derive and implement difference equations for computing a Taylor polynomial approximation to $\sin x$, using the same ideas as in (5.18)–(5.19) for a Taylor polonomial approximation to $e^x$ in Chapter 5.1.7.

The Taylor series for $\sin x$ is presented in Exercise 4.16, Equation (4.19) on page 228. To compute $S(x; n)$ efficiently, we try to compute a new term from the last computed term. Let $S(x; n) = \sum_{j=0}^{n} a_j$, where the expression for a term $a_j$ follows from the formula (4.19). Derive the following relation between two consecutive terms in the series,

$$a_j = -\frac{x^2}{(2j+1)2j}a_{j-1}\,. \tag{5.53}$$

Introduce $s_j = S(x; j-1)$ and define $s_0 = 0$. We use $s_j$ to accumulate terms in the sum. For the first term we have $a_0 = x$. Formulate a system of two difference equations for $s_j$ and $a_j$ in the spirit of (5.18)–(5.19). Implement this system in a function `S(x, n)`, which returns $s_{n+1}$ and $a_{n+1}$. The latter is the first neglected term in the sum (since $s_{n+1} = \sum_{j=0}^{n} a_j$) and may act as a rough measure of the size of the error in the approximation. Suggest how to test that the `S(x, n)` function works correctly. Name of program file: `sin_Taylor_series_diffeq.py`. ◇

**Exercise 5.17.** *Difference equations for computing* $\cos x$*.*

Carry out the steps in Exercise 5.16, but do it for the Taylor series of $\cos x$ instead of $\sin x$ (look up the Taylor series for $\cos x$ in a mathematics textbook or search on the Internet). Name of program file: `cos_Taylor_series_diffeq.py`. ◇

**Exercise 5.18.** *Make a guitar-like sound.*

Given start values $x_0, x_1, \ldots, x_p$, the following diffference equation is known to create guitar-like sound:

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}), \quad n = p+1, \ldots, N\,. \tag{5.54}$$

With a sampling rate $r$, the frequency of this sound is given by $r/p$. Make a program with a function `solve(x, p)` which returns the solution array `x` of (5.54). To initialize the array `x[0:p+1]` we look at two methods, which can be implemented in two alternative functions:

1. $x_0 = 1$, $x_1 = x_2 = \cdots = x_p = 0$
2. $x_0, \ldots, x_p$ are uniformly distributed random numbers in $[-1, 1]$

Import `max_amplitude`, `write`, and `play` from the `scitools.sound` module. Choose a sampling rate $r$ and set $p = r/440$ to create a 440 Hz tone (A). Create an array `x1` of zeros with length $3r$ such that the tone will last for 3 seconds. Initialize `x1` according to method 1 above and solve (5.54). Multiply the `x1` array by `max_amplitude`. Repeat this process for an array `x2` of length $2r$, but use method 2 for the initial values and choose $p$ such that the tone is 392 Hz (G). Concatenate `x1` and `x2`, call `write` and then `play` to play the sound. As you will experience, this sound is amazingly similar to the sound of a guitar string, first playing A for 3 seconds and then playing G for 2 seconds. (The method (5.54) is called the Karplus-Strong algorithm and was discovered in 1979 by a researcher, Kevin Karplus, and his student Alexander Strong, at Stanford University.) Name of program file: `guitar_sound.py`. ◇

**Exercise 5.19.** *Damp the bass in a sound file.*

Given a sequence $x_0,\ldots,x_{N-1}$, the following *filter* transforms the sequence to a new sequence $y_0,\ldots,y_{N-1}$:

$$y_n = \begin{cases} x_n, & n = 0 \\ -\frac{1}{4}(x_{n-1} - 2x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{5.55}$$

If $x_n$ represents sound, $y_n$ is the same sound but with the bass damped. Load some sound file (e.g., the one from Exercise 5.18) or call

```
x = scitools.sound.Nothing_Else_Matters(echo=True)
```

to get a sound sequence. Apply the filter (5.55) and play the resulting sound. Plot the first 300 values in the $x_n$ and $y_n$ signals to see graphically what the filter does with the signal. Name of program file: `damp_bass.py`. ◇

**Exercise 5.20.** *Damp the treble in a sound file.*

Solve Exercise 5.19 to get some experience with coding a filter and trying it out on a sound. The purpose of this exercise is to explore some other filters that reduce the treble instead of the bass. Smoothing the sound signal will in general damp the treble, and smoothing is typically obtained by letting the values in the new filtered sound sequence be an average of the neighboring values in the original sequence.

The simplest smoothing filter can apply a standard average of three neighboring values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{3}(x_{n-1} + x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{5.56}$$

Two other filters put less emphasis on the surrounding values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{5.57}$$

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{16}(x_{n-2} + 4x_{n-1} + 6x_n + 4x_{n+1} + x_{n+2}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{5.58}$$

Apply all these three filters to a sound file and listen to the result. Plot the first 300 values in the $x_n$ and $y_n$ signals for each of the three filters to see graphically what the filter does with the signal. Name of program file: `damp_treble.py`. ◇

**Exercise 5.21.** *Demonstrate oscillatory solutions of (5.13).*

Modify the `growth_logistic.py` program from Chapter 5.1.5 to solve the equation (5.13) on page 244. Read the input parameters $y_0$, $q$, and $N$ from the command line.

Equation (5.13) has the solution $y_n = 1$ as $n \to \infty$. Demonstrate, by running the program, that this is the case when $y_0 = 0.3$, $q = 1$, and $N = 50$.

For larger $q$ values, $y_n$ does not approach a constant limit, but $y_n$ oscillates instead around the limiting value. Such oscillations are sometimes observed in wildlife populations. Demonstrate oscillatory solutions when $q$ is changed to 2 and 3.

It could happen that $y_n$ stabilizes at a constant level for larger $N$. Demonstrate that this is not the case by running the program with $N = 1000$. Name of program file: `growth_logistic2.py`. ◇

**Exercise 5.22.** *Make the program from Exer. 5.21 more flexible.*

It is tedious to run a program like the one from Exercise 5.21 repeatedly for a wide range of input parameters. A better approach is to let the computer do the manual work. Modify the program from Exercise 5.21 such that the computation of $y_n$ and the plot is made in a function. Let the title in the plot contain the parameters $y_0$ and $q$ ($N$ is easily visible from the $x$ axis). Also let the name of the hardcopy reflect the values of $y_0$, $q$, and $N$. Then make loops over $y_0$ and $q$ to perform the following more comprehensive set of experiments:

- $y = 0.01, 0.3$
- $q = 0.1, 1, 1.5, 1.8, 2, 2.5, 3$
- $N = 50$

How does the initial condition (the value $y_0$) seem to influence the solution?

The keyword argument `show=False` can be used in the `plot` call if you do not want all the plot windows to appear on the screen. Name of program file: `growth_logistic3.py`. ◇

**Exercise 5.23.** *Simulate the price of wheat.*

The demand for wheat in year $t$ is given by

$$D_t = ap_t + b,$$

where $a < 0$, $> 0$, and $p_t$ is the price of wheat. Let the supply of wheat be

$$S_t = Ap_{t-1} + B + \ln(1 + p_{t-1}),$$

where $A$ and $B$ are given constants. We assume that the price $p_t$ adjusts such that all the produced wheat is sold. That is, $D_t = S_t$.

For $A = 1$, $a = -3, b = 5, B = 0$, find from numerical computations, a stable price such that the production of wheat from year to year is constant. That is, find $p$ such that $ap + b = Ap + B + \ln(1 + p)$.

Assume that in a very dry year the production of wheat is much less than planned. Given that price this year, $p_0$, is 4.5 and $D_t = S_t$, compute in a program how the prices $p_1, p_2, \ldots, p_N$ develop. This implies solving the difference equation

$$ap_t + b = Ap_{t-1} + B + \ln(1 + p_{t-1}) \,.$$

From the $p_t$ values, compute $S_t$ and plot the points $(p_t, S_t)$ for $t = 0, 1, 2, \ldots, N$. How do the prices move when $N \to \infty$? Name of program file: `wheat.py`. ◇

# 6 Files, Strings, and Dictionaries

Files are used for permanent storage of information on a computer. From previous computer experience you are hopefully used to save information to files and open the files at a later time for inspection again. The present chapter tells you how Python programs can access information in files (Chapter 6.1) and also create new files (Chapter 6.5). The chapter builds on programming concepts introduced in the first four chapters of this book.

Since files often contain structured information that one wants to map to objects in a running program, there is a need for flexible objects where various kinds of other objects can be stored. Dictionaries are very handy for this purpose and are described in Chapter 6.2.

Information in files often appear as pure text, so to interpret and extract data from files it is sometimes necessary to carry out sophisticated operations on the text. Python strings have many methods for performing such operations, and the most important functionality is described in Chapter 6.3.

The World Wide Web is full of information and scientific data that may be useful to access from a program. Chapter 6.4 tells you how to read web pages from a program and interpret the contents using string operations.

The folder `src/files` contains all the program example files referred to in the present chapter.

## 6.1 Reading Data from File

Suppose we have recorded some measurements in a file `data1.txt`, located in the `src/files` folder. The goal of our first example of reading files is to read the measurement values in `data1.txt`, find the average value, and print it out in the terminal window.

Before trying to let a program read a file, we must know the *file format*, i.e., what the contents of the file looks like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file `data1.txt`. To this end, load the file into a text editor or viewer[1]. What we see is a column with numbers:

```
21.8
18.1
19
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items.

### 6.1.1 Reading a File Line by Line

To read a file, we first need to *open* the file. This action creates a file object, here stored in the variable `infile`:

```
infile = open('data1.txt', 'r')
```

The second argument to the `open` function, the string `'r'`, tells that we want to open the file for reading. We shall later see that a file can be opened for writing instead, by providing `'w'` as the second argument. After the file is read, one should close the file object with `infile.close()`.

*For Loop over Lines.* We can read the contents of the file in various ways. The basic recipe for reading the file line by line applies a `for` loop like this:

```
for line in infile:
    # do something with line
```

The `line` variable is a string holding the current line in the file. The `for` loop over lines in a file has the same syntax as when we go through a list. Just think of the file object `infile` as a collection of elements, here lines in a file, and the `for` loop visits these elements in sequence such that the `line` variable refers to one line at a time. If something seemingly goes wrong in such a loop over lines in a file, it is useful to do a `print line` inside the loop.

Instead of reading one line at a time, we can load all lines into a list of strings (lines) by

[1] You can use `emacs`, `vim`, `more`, or `less` on Unix and Mac. On Windows, `WordPad` is appropriate, or the `type` command in a DOS window. Word processors such as OpenOffice or Microsoft Word can also be used.

```
lines = infile.readlines()
```

This statement is equivalent to

```
lines = []
for line in infile:
    lines.append(line)
```

or the list comprehension:

```
lines = [line for line in infile]
```

In the present example, we load the file into the list `lines`. The next task is to compute the average of the numbers in the file. Trying a straightforward sum of all numbers on all lines,

```
mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

gives an error message:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The reason is that `lines` holds each line (`number`) as a string, not a `float` or `int` that we can add to other numbers. A fix is to convert each line to a `float`:

```
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
```

This code snippet works fine. The complete code can be found in the file `files/mean1.py`.

Summing up a list of numbers is often done in numerical programs, so Python has a special function `sum` for performing this task. However, `sum` must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in `lines` into corresponding `float` objects:

```
mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of `float` objects directly. Using this strategy, the complete program (found in file `mean2.py`) takes the form

```
infile = open('data1.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

A newcomer to programming might find it confusing to see that one problem is solved by many alternative sets of statements, but this is the very nature of programming. A clever programmer will judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later. We therefore present more examples on how to read the `data1.txt` file and compute with the data.

*While Loop over Lines.* The call `infile.readline()` returns a string containing the text at the current line. A new `infile.readline()` will read the next line. When `infile.readline()` returns an empty string, the end of the file is reached and we must stop further reading. The following `while` loop reads the file line by line using `infile.readline()`:

```
while True:
    line = infile.readline()
    if not line:
        break
    # process line
```

This is perhaps a somewhat strange loop, but it is a well-established way of reading a file in Python (especially in older codes). The shown `while` loop runs forever since the condition is always `True`. However, inside the loop we test if `line` is `False`, and it is `False` when we reach the end of the file, because `line` then becomes an empty string, which in Python evaluates to `False`. When `line` is `False`, the `break` statement breaks the loop and makes the program flow jump to the first statement after the `while` block.

Computing the average of the numbers in the `data1.txt` file can now be done in yet another way:

```
infile = open('data1.txt', 'r')
mean = 0
n = 0
while True:
    line = infile.readline()
    if not line:
        break
    mean += float(line)
    n += 1
mean = mean/float(n)
```

*Reading a File into a String.* The call `infile.read()` reads the whole file and returns the text as a string object. The following interactive session illustrates the use and result of `infile.read()`:

```
>>> infile = open('data1.txt', 'r')
>>> filestr = infile.read()
>>> filestr
'21.8\n18.1\n19\n23\n26\n17.8\n'
>>> print filestr
21.8
```

```
18.1
19
23
26
17.8
```

Note the difference between just writing `filestr` and writing `print filestr`. The former dumps the string with newlines as "backslash n" characters, while the latter is a "pretty print" where the string is written out without quotes and with the newline characters as visible line shifts[2].

Having the numbers inside a string instead of inside a file does not look like a major step forward. However, string objects have many useful functions for extracting information. A very useful feature is *split*: `filestr.split()` will split the string into words (separated by blanks or any other sequence of characters you have defined). The "words" in this file are the numbers:

```
>>> words = filestr.split()
>>> words
['21.8', '18.1', '19', '23', '26', '17.8']
>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)
>>> print mean
20.95
```

A more compact program looks as follows (`mean3.py`):

```
infile = open('data1.txt', 'r')
numbers = [float(w) for w in infile.read().split()]
mean = sum(numbers)/len(numbers)
```

The next section tells you more about splitting strings.

### 6.1.2 Reading a Mixture of Text and Numbers

The `data1.txt` file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file `rainfall.dat` provides an example[3]:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan  81.2
Feb  63.2
Mar  70.3
Apr  55.7
May  53.0
Jun  36.4
Jul  17.5
Aug  27.5
Sep  60.9
Oct  117.7
Nov  111.0
Dec  97.9
Year 792.9
```

[2] The difference between these two outputs is explained in Chapter 7.3.9.

[3] `http://www.worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1`

How can we read the rainfall data in this file and make a plot of the values?

The most straightforward solution is to read the file line by line, and for each line split the line into words, pick out the last (second) word on the line, convert this word to `float`, and store the `float` objects in a list. Having the rainfall values in a list of real numbers, we can make a plot of the values versus the month number. The complete code, wrapped in a function, may look like this (file `rainfall.py`):

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    numbers = []
    for line in infile:
        words = line.split()
        number = float(words[1])
        numbers.append(number)
    infile.close()
    return numbers

values = extract_data('rainfall.dat')
from scitools.std import plot
month_indices = range(1, 13)
plot(month_indices, values[:-1], 'o2')
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by `infile.readline()`. The `for` loop over the lines in the file will then start from the next (second) line.

Also note that `numbers` contain data for the 12 months plus the average annual rainfall. We want to plot the average rainfall for the months only, i.e., `values[0:12]` or simply `values[:-1]` (everything except the last entry). Along the "x" axis we put the index of a month, starting with 1. A call to `range(1,13)` generates these indices.

We can condense the `for` loop over lines in the file, if desired, by using a list comprehension:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline()  # skip the first line
    numbers = [float(line.split()[1]) for line in infile]
    infile.close()
    return numbers
```

### 6.1.3 What Is a File, Really?

This section is not mandatory for understanding the rest of the book. However, we think the information here is fundamental for understanding what files are about.

A file is simply a sequence of characters. In addition to the sequence of characters, a file has some data associated with it, typically the

name of the file, its location on the disk, and the file size. These data are stored somewhere by the operating system. Without this extra information beyond the pure file contents as a sequence of characters, the operating system cannot find a file with a given name on the disk.

Each character in the file is represented as a *byte*, consisting of eight *bits*. Each bit is either 0 or 1. The zeros and ones in a byte can be combined in $2^8 = 256$ ways. This means that there are 256 different types of characters. Some of these characters can be recognized from the keyboard, but there are also characters that do not have a familiar symbol. The name of such characters looks cryptic when printed.

*Pure Text Files.* To see that a file is really just a sequence of characters, invoke an editor for plain text, e.g., the editor you use to write Python programs. Write the four characters `ABCD` into the editor, do not press the Return key, and save the text to a file `test1.txt`. Use your favorite tool for file and folder overview and move to the folder containing the `test1.txt` file. This tool may be Windows Explorer, My Computer, or a DOS window on Windows; a terminal window, Konqueror, or Nautilus on Linux; or a terminal window or Finder on Mac. If you choose a terminal window, use the `cd` (change directory) command to move to the proper folder and write `dir` (Windows) or `ls -l` (Linux/Mac) to list the files and their sizes. In a graphical program like Windows Explorer, Konqueror, Nautilus, or Finder, select a view that shows the *size* of each file[4]. You will see that the `test1.txt` file has a size of 4 bytes[5]. The 4 bytes are exactly the 4 characters `ABCD` in the file. Physically, the file is just a sequence of 4 bytes on your harddisk.

Go back to the editor again and add a newline by pressing the Return key. Save this new version of the file as `test2.txt`. When you now check the size of the file it has grown to five bytes. The reason is that we added a newline character (symbolically known as "backslash n").

Instead of examining files via editors and folder viewers we may use Python interactively:

```
>>> file1 = open('test1.txt', 'r').read()  # read file into string
>>> file1
'ABCD'
>>> len(file1)        # length of string in bytes/characters
4
>>> file2 = open('test2.txt', 'r').read()
>>> file2
'ABCD\n'
>>> len(file2)
5
```

Python has in fact a function that returns the size of a file directly:

[4] Choose "view as details" in Windows Explorer, "View as List" in Nautilus, the list view icon in Finder, or you just point at a file icon in Konqueror and watch the pop-up text.

[5] If you use `ls -l`, the size measured in bytes is found in column 5, right before the date.

```
>>> import os
>>> size = os.path.getsize('test1.txt')
>>> size
4
```

*Word Processor Files.* Most computer users write text in a word processing program, such as Microsoft Word or OpenOffice. Let us investigate what happens with our four characters `ABCD` in such a program. Start the word processor, open a new document, and type in the four characters `ABCD` only. Save the document as a `.doc` file (Microsoft Word) or an `.odt` file (OpenOffice). Load this file into an editor for pure text and look at the contents. You will see that there are numerous strange characters that you did not write (!). This additional "text" contains information on what type of document this is, the font you used, etc. The OpenOffice version of this file has 5725 bytes! However, if you save the file as a pure text file, with extention `.txt`, the size is not more than four bytes, and the text file contains just the corresponding characters `ABCD`.

Instead of loading the OpenOffice file into an editor we can again read the file contents into a string in Python and examine this string:

```
>>> infile = open('test3.odt', 'r')  # open OpenOffice file
>>> s = infile.read()
>>> len(s)   # file size
5725
>>> s
'PK\x03\x04\x14\x00\x00\x00\x00\x00r\x80E6^\xc62\x0c\...
\x00\x00mimetypeapplication/vnd.oasis.opendocument.textPK\x00...
\x00\x00content.xml\xa5VMS\xdb0\x10\xbd\xf7Wx|\xe8\xcd\x11...'
```

Each backslash followed by `x` and a number is a code for a special character not found on the keyboard (recall that there are 256 characters and only a subset is associated with keyboard symbols). Although we show just a small portion of all the characters in this file in the above output[6], we can guarantee that you cannot find the pure sequence of characters `ABCD`. However, the computer program that generated the file, OpenOffice in this example, can easily interpret the meaning of all the characters in the file and translate the information into nice, readable text on the screen.

*Image Files.* A digital image – captured by a digital camera or a mobile phone – is a file. And since it is a file, the image is just a sequence of characters. Loading some JPEG file into a pure text editor, you can see all the strange characters in there. On the first line you will (normally) find some recognizable text in between the strange characters. This text reflects the type of camera used to capture the image and the date and time when the picture was taken. The next lines contain

[6] Otherwise, the output would have occupied several pages in this book with about five thousand backslash-x-number symbols...

more information about the image. Thereafter, the file contains a set of numbers representing the image. The basic representation of an image is a set of $m \times n$ pixels, where each pixel has a color represented as a combination of 256 values of red, green, and blue. A 6 megapixel camera will then need to store $256 \times 3$ bytes for each of the 6,000,000 pixels, which results in $4,608,000,000$ bytes (or 4.6 gigabytes, written 4.6 Gb). The JPEG file contains only a couple of megabytes. The reason is that JPEG is a *compressed* file format, produced by applying a smart technique that can throw away pixel information in the original picture such that the human eye hardly can detect the inferior quality.

A video is just a sequence of images, and therefore a video is also a stream of bytes. If the change from one video frame (image) to the next is small, one can use smart methods to compress the image information in time. Such compression is particularly important for videos since the file sizes soon get too large for being transferred over the Internet. A small video file occasionally has bad visual quality, caused by too much compression.

*Music Files.* An MP3 file is much like a JPEG file: First, there is some information about the music (artist, title, album, etc.), and then comes the music itself as a stream of bytes. A typical MP3 file has a size of something like five million bytes[7], i.e., five megabytes (5 Mb). On a 2 Gb MP3 player you can then store roughly $2,000,000,000/5,000,000 = 400$ MP3 files. MP3 is, like JPEG, a compressed format. The complete data of a song on a CD (the WAV file) contains about ten times as many bytes. As for pictures, the idea is that one can throw away a lot of bytes in an intelligent way, such that the human ear hardly detects the difference between a compressed and uncompressed version of the music file.

*PDF Files.* Looking at a PDF file in a pure text editor shows that the file contains some readable text mixed with some unreadable characters. It is not possible for a human to look at the stream of bytes and deduce the text in the document[8]. A PDF file reader can easily interpret the contents of the file and display the text in a human-readable form on the screen.

*Remarks.* We have repeated many times that a file is just a stream of bytes. A human can interpret (read) the stream of bytes if it makes sense in a human language – or a computer language (provided the human is a programmer). When the series of bytes does not make

[7] The exact size depends on the complexity of the music, the length of the track, and the MP3 resolution.

[8] From the assumption that there are always some strange people doing strange things, there might be somebody out there who – with a lot of training – can interpret the pure PDF code with the eyes.

sense to any human, a computer program must be used to interpret the sequence of characters.

Think of a report. When you write the report as pure text in a text editor, the resulting file contains just the characters you typed in from the keyboard. On the other hand, if you applied a word processor like Microsoft Word or OpenOffice, the report file contains a large number of extra bytes describing properties of the formatting of the text. This stream of extra bytes does not make sense to a human, and a computer program is required to interpret the file content and display it in a form that a human can understand. Behind the sequence of bytes in the file there are strict rules telling what the series of bytes means. These rules reflect the *file format*. When the rules or file format is publicly documented, a programmer can use this documentation to make her own program for interpreting the file contents[9]. It happens, though, that secret file formats are used, which require certain programs from certain companies to interpret the files.

## 6.2 Dictionaries

So far in the book we have stored information in various types of objects, such as numbers, strings, list, and arrays. A *dictionary* is a very flexible object for storing various kind of information, and in particular when reading files. It is therefore time to introduce the dictionary type.

A list is a collection of objects indexed by an integer going from 0 to the number of elements minus one. Instead of looking up an element through an integer index, it can be more handy to use a text. Roughly speaking, a list where the index can be a text is called a dictionary in Python. Other computer languages use other names for the same thing: HashMap, hash, associative array, or map.

### 6.2.1 Making Dictionaries

Suppose we need to store the temperatures from three cities: Oslo, London, and Paris. For this purpose we can use a list,

```
temps = [13, 15.4, 17.5]
```

but then we need to remember the sequence of cities, e.g., that index 0 corresponds to Oslo, index 1 to London, and index 2 to Paris. That is, the London temperature is obtained as `temps[1]`. A dictionary with the city name as index is more convenient, because this allows us to

[9] Interpreting such files is much more complicated than our examples on reading human-readable files in this book.

write temps['London'] to look up the temperature in London. Such a dictionary is created by one of the following two statements

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
# or
temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Additional text-value pairs can be added when desired. We can, for instance, write

```
temps['Madrid'] = 26.0
```

The temps dictionary has now four text-value pairs, and a print temps yields

```
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5, 'Madrid': 26.0}
```

### 6.2.2 Dictionary Operations

The string "indices" in a dictionary are called *keys*. To loop over the keys in a dictionary d, one writes for key in d: and works with key and the corresponding value d[key] inside the loop. We may apply this technique to write out the temperatures in the temps dictionary from the previous paragraph:

```
>>> for city in temps:
...     print 'The temperature in %s is %g' % (city, temps[city])
...
The temperature in Paris is 17.5
The temperature in Oslo is 13
The temperature in London is 15.4
The temperature in Madrid is 26
```

We can check if a key is present in a dictionary by the syntax if key in d:

```
>>> if 'Berlin' in temps:
...     print 'Berlin:', temps['Berlin']
... else:
...     print 'No temperature data for Berlin'
...
No temperature data for Berlin
```

Writing key in d yields a standard boolean expression, e.g.,

```
>>> 'Oslo' in temps
True
```

The keys and values can be extracted as lists from a dictionary:

```
>>> temps.keys()
['Paris', 'Oslo', 'London', 'Madrid']
>>> temps.values()
[17.5, 13, 15.4, 26.0]
```

An important feature of the `keys` method in dictionaries is that the order of the returned list of keys is unpredictable. If you need to traverse the keys in a certain order, you will need to sort the keys. A loop over the keys in the `temps` dictionary in alphabetic order is written as

```
>>> for city in sorted(temps):
...     print city
...
London
Madrid
Oslo
Paris
```

A key-value pair can be removed by `del d[key]`:

```
>>> del temps['Oslo']
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps)  # no of key-value pairs in dictionary
3
```

Sometimes we need to take a copy of a dictionary:

```
>>> temps_copy = temps.copy()
>>> del temps_copy['Paris']   # this does not affect temps
>>> temps_copy
{'London': 15.4, 'Madrid': 26.0}
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

Note that if two variables refer to the same dictionary and we change the contents of the dictionary through either of the variables, the change will be seen in both variables:

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0    # change t1
>>> temps                     # temps is also changed
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

To avoid that `temps` is affected by adding a new key-value pair to `t1`, `t1` must be a copy of `temps`.

### 6.2.3 Example: Polynomials as Dictionaries

The keys in a dictionary are not restricted to be strings. In fact, any Python object whose contents cannot be changed can be used as key[10]. For example, we may use integers as keys in a dictionary. This is a handy way of representing polynomials, as will be explained next.

Consider the polynomial

[10] Such objects are known as *immutable* data types and consist of `int`, `float`, `complex`, `str`, and `tuple`. Lists and dictionaries can change their contents and are called *mutable* objects. These cannot be used as keys in dictionaries. If you desire a list as key, use a tuple instead.

$$p(x) = -1 + x^2 + 3x^7 \,.$$

The data associated with this polynomial can be viewed as a set of power-coefficient pairs, in this case the coefficient $-1$ belongs to power 0, the coefficient 1 belongs to power 2, and the coefficient 3 belongs to power 7. A dictionary can be used to map a power to a coefficient:

```
p = {0: -1, 2: 1, 7: 3}
```

A list can, of course, also be used, but in this case we must fill in all the zero coefficients too, since the index must match the power:

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

The advantage with a dictionary is that we need to store only the non-zero coefficients. For the polynomial $1 + x^{100}$ the dictionary holds two elements while the list holds 101 elements (see Exercise 6.16).

The following function can be used to evaluate a polynomial represented as a dictionary:

```
def poly1(data, x):
    sum = 0.0
    for power in data:
        sum += data[power]*x**power
    return sum
```

The `data` argument must be a dictionary where `data[power]` holds the coefficient associated with the term `x**power`. A more compact implementation can make use of Python's `sum` function to sum the elements of a list:

```
def poly1(data, x):
    return sum([data[p]*x**p for p in data])
```

That is, we first make a list of the terms in the polynomial using a list comprehension, and then we feed this list to the `sum` function. Note that the name `sum` is different in the two implementations: In the first, `sum` is a `float` object, and in the second, `sum` is a function. When we set `sum=0.0` in the first implementation, we bind the name `sum` to a new `float` object, and the built-in Python function associated with the name `sum` is then no longer accessible inside the `poly1` function[11]. Outside the function, nevertheless, `sum` will be the summation function (unless we have bound the global name `sum` to another object somewhere else in the main program – see Chapter 2.4.2 for a discussion of this issue).

With a list instead of dictionary for representing the polynomial, a slightly different evaluation function is needed:

[11] This is not strictly correct, because `sum` is a local variable while the summation function is associated with a global name `sum`, which can be reached through `globals()['sum']`.

```
def poly2(data, x):
    sum = 0
    for power in range(len(data)):
        sum += data[power]*x**power
    return sum
```

If there are many zeros in the `data` list, `poly2` must perform all the multiplications with the zeros, while `poly1` computes with the non-zero coefficients only and is hence more efficient.

Another major advantage of using a dictionary to represent a polynomial rather than a list is that negative powers are easily allowed, e.g.,

```
p = {-3: 0.5, 4: 2}
```

can represent $\frac{1}{2}x^{-3} + 2x^4$. With a list representation, negative powers require much more book-keeping. We may, for example, set

```
p = [0.5, 0, 0, 0, 0, 0, 0, 2]
```

and remember that `p[i]` is the coefficient associated with the power `i-3`. In particular, the `poly2` function will no longer work for such lists, while the `poly1` function works also for dictionaries with negative keys (powers).

You are now encouraged to solve Exercise 6.17 on page 327 to become more familiar with the concept of dictionaries.

### 6.2.4 Example: File Data in Dictionaries

*Problem.* The file `files/densities.dat` contains a table of densities of various substances measured in g/cm$^3$:

```
air         0.0012
gasoline    0.67
ice         0.9
pure water  1.0
seawater    1.025
human body  1.03
limestone   2.6
granite     2.7
iron        7.8
silver      10.5
mercury     13.6
gold        18.9
platinium   21.4
Earth mean  5.52
Earth core  13
Moon        3.3
Sun mean    1.4
Sun core    160
proton      2.8E+14
```

In a program we want to access these density data. A dictionary with the name of the substance as key and the corresponding density as value seems well suited for storing the data.

*Solution.* We can read the `densities.dat` file line by line, split each line into words, use a float conversion of the last word as density value, and the remaining one or two words as key in the dictionary.

```
def read_densities(filename):
    infile = open(filename, 'r')
    densities = {}
    for line in infile:
        words = line.split()
        density = float(words[-1])

        if len(words[:-1]) == 2:
            substance = words[0] + ' ' + words[1]
        else:
            substance = words[0]

        densities[substance] = density
    infile.close()
    return densities

densities = read_densities('densities.dat')
```

This code is found in the file `density.py`. With string operations from Chapter 6.3.1 we can avoid the special treatment of one or two words in the name of the substance and achieve simpler and more general code, see Exercise 6.10.

### 6.2.5 Example: File Data in Nested Dictionaries

*Problem.* We are given a data file with measurements of some properties with given names (here `A`, `B`, `C` ...). Each property is measured a given number of times. The data are organized as a table where the rows contain the measurements and the columns represent the measured properties:

```
        A       B       C      D
1     11.7    0.035    2017    99.1
2      9.2    0.037    2019   101.2
3     12.2     no       no    105.2
4     10.1    0.031     no    102.1
5      9.1    0.033    2009   103.3
6      8.7    0.036    2015   101.9
```

The word "no" stands for no data, i.e., we lack a measurement. We want to read this table into a dictionary `data` so that we can look up measurement no. `i` of (say) property `C` as `data['C'][i]`. For each property `p`, we want to compute the mean of all measurements and store this as `data[p]['mean']`.

*Algorithm.* The algorithm for creating the `data` dictionary goes as follows:

```
examine the first line: split it into words and
            initialize a dictionary with the property names
            as keys and empty dictionaries ({}) as values
for each of the remaining lines in the file:
    split the line into words
    for each word after the first:
        if the word is not "no":
            transform the word to a real number and store
                    the number in the relevant dictionary
```

*Implementation.* The solution requires familiarity with dictionaries and list slices (also called sublists, see Chapter 2.1.9). A new aspect needed in the solution is *nested dictionaries*, that is, dictionaries of dictionaries. The latter topic is first explained, via an example:

```
>>> d = {'key1': {'key1': 2, 'key2': 3}, 'key2': 7}
```

Observe here that the value of `d['key1']` is a dictionary which we can index with its keyes `key1` and `key2`:

```
>>> d['key1']           # this is a dictionary
{'key2': 3, 'key1': 2}
>>> type(d['key1'])     # proof
<type 'dict'>
>>> d['key1']['key1']   # index a nested dictionary
2
>>> d['key1']['key2']
3
```

In other words, repeated indexing works for nested dictionaries as for nested lists. The repeated indexing does not apply to `d['key2']` since that value is just an integer:

```
>>> d['key2']['key1']
  ...
TypeError: unsubscriptable object
>>> type(d['key2'])
<type 'int'>
```

When we have understood the concept of nested dictionaries, we are in a position to present a complete code that solves our problem of loading the tabular data in the file `table.dat` into a nested dictionary `data` and computing mean values. First, we list the program, stored in the file `table2dict.py`, and display the program's output. Thereafter, we dissect the code in detail.

```
infile = open('table.dat', 'r')
lines = infile.readlines()
infile.close()
data = {}   #  data[property][measurement_no] = propertyvalue
first_line = lines[0]
properties = first_line.split()
for p in properties:
```

```
    data[p] = {}

for line in lines[1:]:
    words = line.split()
    i = int(words[0])       # measurement number
    values = words[1:]      # values of properties
    for p, v in zip(properties, values):
        if v != 'no':
            data[p][i] = float(v)

# compute mean values:
for p in data:
    values = data[p].values()
    data[p]['mean'] = sum(values)/len(values)

for p in sorted(data):
    print 'Mean value of property %s = %g' % (p, data[p]['mean'])
```

The corresponding output from this program becomes

```
Mean value of property A = 9.875
Mean value of property B = 0.0279167
Mean value of property C = 1678.42
Mean value of property D = 98.0417
```

To view the nested `data` dictionary, we may insert

```
import scitools.pprint2; scitools.pprint2.pprint(temps)
```

which produces something like

```
{'A': {1: 11.7, 2: 9.2, 3: 12.2, 4: 10.1, 5: 9.1, 6: 8.7,
       'mean': 10.1667},
 'B': {1: 0.035, 2: 0.037, 4: 0.031, 5: 0.033, 6: 0.036,
       'mean': 0.0344},
 'C': {1: 2017, 2: 2019, 5: 2009, 6: 2015, 'mean': 2015},
 'D': {1: 99.1,
       2: 101.2,
       3: 105.2,
       4: 102.1,
       5: 103.3,
       6: 101.9,
       'mean': 102.133}}
```

*Dissection.* To understand a computer program, you need to understand what the result of every statement is. Let us work through the code, almost line by line, and see what it does.

First, we load all the lines of the file into a list of strings called `lines`. The `first_line` variable refers to the string

```
'       A        B       C      D'
```

We split this line into a list of words, called `properties`, which then contains

```
['A', 'B', 'C', 'D']
```

With each of these property names we associate a dictionary with the measurement number as key and the property value as value, but first we must create these "inner" dictionaries as empty before we can add the measurements:

```
for p in properties:
    data[p] = {}
```

The first pass in the `for` loop picks out the string

```
'1    11.7    0.035    2017    99.1'
```

as the `line` variable. We split this line into words, the first word (`words[0]`) is the measurement number, while the rest `words[1:]` is a list of property values, here named `values`. To pair up the right properties and values, we loop over the `properties` and `values` lists simultaneously:

```
    for p, v in zip(properties, values):
        if v != 'no':
            data[p][i] = float(v)
```

Recall that some values may be missing and we drop to record that value[12]. Because the `values` list contains strings (words) read from the file, we need to explicitly transform each string to a `float` number before we can compute with the values.

After the `for line in lines[1:]` loop, we have a dictionary `data` of dictionaries where all the property values are stored for each measurement number and property name. Figure 6.1 shows a graphical representation of the `data` dictionary.

It remains to compute the average values. For each property name `p`, i.e., key in the `data` dictionary, we can extract the recorded values as the list `data[p].values()` and simply send this list to Python's `sum` function and divide by the number of measured values for this property, i.e., the length of the list:

```
for p in data:
    values = data[p].values()
    data[p]['mean'] = sum(values)/len(values)
```

Alternatively, we can write an explicit loop to compute the average:

```
for p in data:
    sum_values = 0
    for value in data[p]:
        sum_values += value
    data[p]['mean'] = sum_values/len(data[p])
```

When we want to look up a measurement no. `n` of property `B`, we must recall that this particular measurement may be missing so we must do a test if `n` is key in the dictionary `data[p]`:

```
if n in data['B']:
    value = data['B'][n]

# alternative:
value = data['B'][n] if n in data['B'] else None
```

[12] We could, alternatively, set the value to `None`.

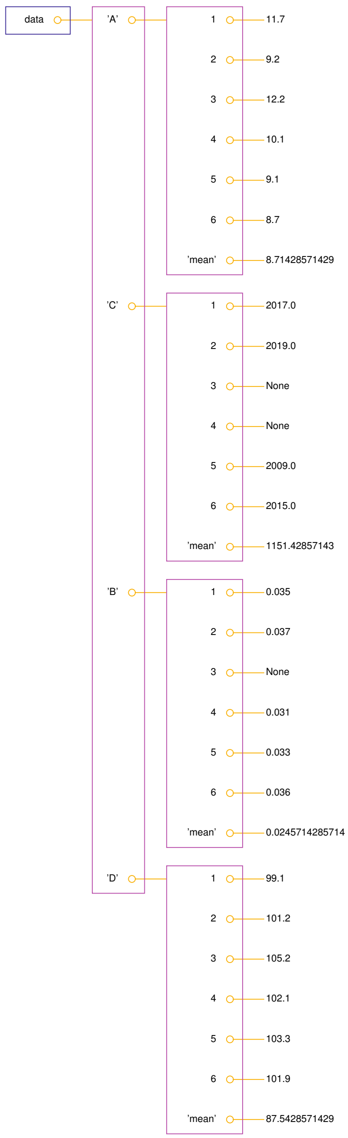



**Fig. 6.1** Illustration of the nested dictionary created in the `table2dict.py` program.

### 6.2.6 Example: Comparing Stock Prices

*Problem.* We want to compare the evolution of the stock prices of three giant companies in the computer industry: Microsoft, Sun Microsystems, and Google. Relevant data files for stock prices can be downloaded from *finance.yahoo.com*. Fill in the comany's name and click on "GET QUOTES" in the top bar of this page, then choose "Historical Prices". On the resulting web page we can specify start and end dates for the historical prices of the stock. We let this be January 1, 1988, for Microsoft and Sun, and January 1, 2005, for Google. The end dates were set to June 1, 2008, in this example. Ticking off "Monthly"

values and clicking "Get Prices" result in a table of stock prices. We can download the data in a tabular format by clicking "Download To Spreadsheet" below the table. Here is an example of such a file:

```
Date,Open,High,Low,Close,Volume,Adj Close
2008-06-02,12.91,13.06,10.76,10.88,16945700,10.88
2008-05-01,15.50,16.37,12.37,12.95,26140700,12.95
2008-04-01,15.78,16.23,14.62,15.66,10330100,15.66
2008-03-03,16.35,17.38,15.41,15.53,12238800,15.53
2008-02-01,17.47,18.03,16.06,16.40,12147900,16.40
2008-01-02,17.98,18.14,14.20,17.50,15156100,17.50
2007-12-03,20.61,21.55,17.96,18.13,9869900,18.13
2007-11-01,5.65,21.60,5.10,20.78,17081500,20.78
```

The file format is simple: columns are separated by comma, the first line contains column headings, and the data lines have the date in the first column and various measures of stock prices in the next columns. Reading about the meaning of the various data on the Yahoo! web pages reveals that our interest concerns the final column (these prices are adjusted for splits and dividends). Three relevant data files can be found in `src/files` with the names `company_monthly.csv`, where `company` is `Microsoft`, `Sun`, or `Google`.

The task is to plot the evolution of stock prices of the three companies. It is natural to scale the prices to start at a unit value in January 1988 and let the Google price start at the maximum of the Sun and Microsoft stock values in January 2005.

*Solution.* There are two major parts of this problem: (i) reading the file and (ii) plotting the data. The reading part is quite straightforward, while the plotting part needs some special considerations since the "x" values in the plot are dates and not real numbers. In the forthcoming text we solve the individual subproblems one by one, showing the relevant Python snippets. The complete program is found in the file `stockprices.py`.

We start with the reading part. Since the reading will be repeated for three files, we make a function with the filename as argument. The result of reading a file should be two lists (or arrays) with the dates and the stock prices, respectively. We therefore return these two lists from the function. The algorithm for reading the data goes as follows:

```
open the file
create two empty lists, dates and prices, for collecting the data
read the first line (of no interest)
for each line in the rest of the file:
    split the line wrt. colon
    append the first word on the line to the dates list
    append the last word on the line to the prices list
close the file
```

There are a couple of additional points to consider. First, the words on a line are strings, and at least the prices (last word) should be

converted to a float. The first word, the date, has the form year-month-day (e.g., 2008-02-04). Since we asked for monthly data only, the day part is of no interest. Skipping the day part can be done by extracting a substring of the date string: `date[:-3]`, which means everything in the string except the last three characters (see Chapter 6.3.1 for more on substrings). The remaining date specification is now of the form year-month (e.g., 2008-02), represented as a string. Turning this into a number for plotting is not so easy, so we keep this string as it is in the list of dates.

The second point of consideration in the algorithm above is the sequence of data in the two lists: the files have the most recent date at the top and the oldest at the bottom, while it is natural to plot the evolution of stock prices against *increasing* time. Therefore, we must reverse the two lists of data before we return them to the calling code.

The algorithm above, together with the two additional comments, can now be translated into Python code:

```
def read_file(filename):
    infile = open(filename, 'r')
    infile.readline()  # read column headings
    dates = [];  prices = []
    for line in infile:
        columns = line.split(',')
        date = columns[0]
        date = date[:-3]  # skip day of month
        price = columns[-1]
        dates.append(date)
        prices.append(float(price))
    infile.close()
    dates.reverse()
    prices.reverse()
    return dates, prices
```

The reading of a file is done by a call to this function, e.g.,

```
dates_Google, prices_Google = read_file('stockprices_Google.csv')
```

Instead of working with separate variables for the file data, we may collect the data in dictionaries, with the company name as key. One possibility is to use two dictionaries:

```
dates = {};  prices = {}
d, p = read_file('stockprices_Sun.csv')
dates['Sun'] = d;  prices['Sun'] = p
d, p = read_file('stockprices_Microsoft.csv')
dates['MS'] = d;  prices['MS'] = p
d, p = read_file('stockprices_Google.csv')
dates['Google'] = d;  prices['Google'] = p
```

We can also collect the `dates` and `prices` dictionaries in a dictionary `data`:

```
data = {'prices': prices, 'dates': dates}
```

Note that `data` is a nested dictionary, so that to extract, e.g., the prices of the Microsoft stock, one writes `data['prices']['MS']`.

The next task is to normalize the stock prices so that we can easily compare them. The idea is to let Sun and Microsoft start out with a unit price and let Google start out with the best of the Sun and Microsoft prices. Normalizing the Sun and Microsoft prices is done by dividing by the first prices:

```
norm_price = prices['Sun'][0]
prices['Sun'] = [p/norm_price for p in prices['Sun']]
```

with a similar code for the Microsoft prices. Normalizing the Google prices is more involved as we need to extract the prices of Sun and Microsoft stocks from January 2005. Since the `dates` and `prices` lists correspond to each other, element by element, we can get the index corresponding to the date `'2005-01'` in the list of dates and use this index to extract the corresponding price. The normalization can then be coded as

```
jan05_MS = prices['MS'][dates['MS'].index('2005-01')]
jan05_Sun = prices['Sun'][dates['Sun'].index('2005-01')]
norm_price = prices['Google'][0]/max(jan05_MS, jan05_Sun)
prices['Google'] = [p/norm_price for p in prices['Google']]
```

The purpose of the final plot is to show how the prices evolve in time. The problem is that our time data consists of strings of the form year-month. We need to convert this string information to some "x" coordinate information in the plot. The simplest strategy is to just plot the prices against the list index, i.e., the "x" coordinates correspond to counting months. Suitable lists of monthly based indices for Sun and Microsoft are straightforward to create with the `range` function:

```
x = {}
x['Sun'] = range(len(prices['Sun']))
x['MS']  = range(len(prices['MS']))
```

The "x" coordinates for the Google prices are somewhat more complicated, because the indices must start at the index corresponding to January 2005 in the Sun and Microsoft data. However, we extracted that index in the normalization of the Google prices, so we have already done most of the work:

```
jan05 = dates['Sun'].index('2005-01')
x['Google'] = range(jan05, jan05 + len(prices['Google']), 1)
```

The final step is to plot the three set of data:

```
from scitools.std import plot
plot(x['MS'], prices['MS'], 'r-',
     x['Sun'], prices['Sun'], 'b-',
     x['Google'], prices['Google'], 'y-',
     legend=('Microsoft', 'Sun', 'Google'))
```

Figure 6.2 displays the resulting plot. As seen from the plot, the best investment would be to start with Microsoft stocks in 1988 and switch all the money to Google stocks in 2005. You can easily modify the program to explores what would happen if you started out with Sun stocks and switched to Google in 2005.

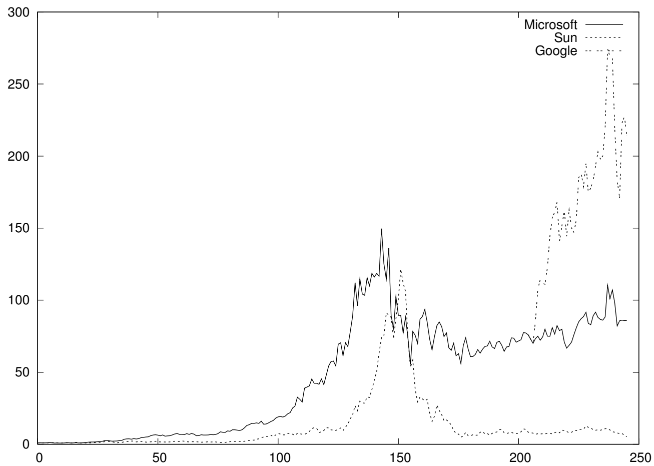



**Fig. 6.2** The evolution of stock prices for three companies in the period January 1998 to June 2008.

*Generalization.* We can quite easily generalize the program to handle data from an arbitrary collection of companies, at least if we restrict the time period to be the same for all stocks. Exercise 6.18 asks you to do this. As you will realize, the use of dictionaries instead of separate variables in our program consitutes one important reason why the program becomes easy to extend. Avoiding different time periods for different price data also makes the generalized program simpler than the one we developed above.

## 6.3 Strings

Many programs need to manipulate text. For example, when we read the contents of a file into a string or list of strings (lines), we may want to change parts of the text in the string(s) – and maybe write out the modified text to a new file. So far in this chapter we have converted parts of the text to numbers and computed with the numbers. Now it is time to learn how to manipulate the text strings themselves.

### 6.3.1 Common Operations on Strings

Python has a rich set of operations on string objects. Some of the most common operations are listed below.

*Substring Specification.* The expression `s[i:j]` extracts the substring starting with character number `i` and ending with character number `j-1` (similarly to lists, `0` is the index of the first character):

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s[8:]     # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]   # index 8, 9, 10 and 11 (not 12!)
'18.4'
```

A negative upper index counts, as usual, from the right such that `s[-1]` is the last element, `s[-2]` is the next last element, and so on.

```
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

*Searching for Substrings.* The call `s.find(s1)` returns the index where the substring `s1` first appears in `s`. If the substring is not found, -1 is returned.

```
>>> s.find('Berlin')  # where does 'Berlin' start?
0
>>> s.find('pm')
20
>>> s.find('Oslo')    # not found
-1
```

Sometimes the aim is to just check if a string is contained in another string, and then we can use the syntax:

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False
```

Here is a typical use of the latter construction in an `if` test:

```
>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
...
C found
```

Two other convenient methods for checking if a string starts with or ends with a specified string are `startswith` and `endswith`:

```
>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False
```

*Substitution.* The call `s.replace(s1, s2)` replaces substring `s1` by `s2` everywhere in `s`:

```
>>> s.replace(' ', '_')
'Berlin:_18.4_C__at_4_pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

A variant of the last example, where several string operations are put together, consists of replacing the text before the first colon[13]:

```
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

*String Splitting.* The call `s.split()` splits the string `s` into words separated by whitespace (space, tabulator, or newline):

```
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

Splitting a string `s` into words separated by a text `t` can be done by `s.split(t)`. For example, we may split with respect to colon:

```
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
```

We know that `s` contains a city name, a colon, a temperature, and then `C`:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
```

With `s.splitlines()`, a multi-line string is split into lines (very useful when a file has been read into a string and we want a list of lines):

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
2nd line
3rd line
>>> t.splitlines()
['1st line', '2nd line', '3rd line']
```

[13] Take a "break" and convince yourself that you understand how we specify the substring to be replaced.

*Upper and Lower Case.* `s.lower()` transforms all characters to their lower case equivalents, and `s.upper()` performs a similar transformation to upper case letters:

```
>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'
```

*Strings Are Constant.* A string cannot be changed, i.e., any change always results in a new string. Replacement of a character is not possible:

```
>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment
```

If we want to replace `s[18]`, a new string must be constructed, for example by keeping the substrings on either side of `s[18]` and inserting a `'5'` in between:

```
>>> s[:18] + '5' + s[19:]
'Berlin: 18.4 C at 5 pm'
```

*Strings with Digits Only.* One can easily test whether a string contains digits only or not:

```
>>> '214'.isdigit()
True
>>> '  214 '.isdigit()
False
>>> '2.14'.isdigit()
False
```

*Whitespace.* We can also check if a string contains spaces only by calling the `isspace` method. More precisely, `isspace` tests for *whitespace*, which means the space character, newline, or the TAB character:

```
>>> '    '.isspace()   # blanks
True
>>> '  \n'.isspace()   # newline
True
>>> '  \t '.isspace()  # TAB
True
>>> ''.isspace()       # empty string
False
```

The `isspace` is handy for testing for blank lines in files. An alternative is to strip first and then test for an empty string:

```
>>> line = '   \n'
>>> empty.strip() == ''
True
```

Stripping off leading and/or trailing spaces in a string is sometimes useful:

```
>>> s = '   text with leading/trailing space   \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip()   # left strip
'text with leading/trailing space   \n'
>>> s.rstrip()   # right strip
'   text with leading/trailing space'
```

*Joining Strings.* The opposite of the `split` method is `join`, which joins elements in a list of strings with a specified delimiter in between. That is, the following two types of statements are inverse operations:

```
t = delimiter.join(words)
words = t.split(delimiter)
```

An example on using `join` may be

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> t = ', '.join(strings)
>>> t
'Newton, Secant, Bisection'
```

As an illustration of the usefulness of `split` and `join`, we want to remove the first two words on a line. This task can be done by first splitting the line into words and then joining the words of interest:

```
>>> line = 'This is a line of words separated by space'
>>> words = line.split()
>>> line2 = ' '.join(words[2:])
>>> line2
'a line of words separated by space'
```

There are many more methods in string objects. All methods are described in the Python Library Reference, see “string methods” in the index.

### 6.3.2 Example: Reading Pairs of Numbers

*Problem.* Suppose we have a file consisting of pairs of real numbers, i.e., text of the form $(a, b)$, where $a$ and $b$ are real numbers. This notation for a pair of numbers is often used for points in the plane, vectors in the plane, and complex numbers. A sample file may look as follows:

```
(1.3,0)    (-1,2)    (3,-1.5)
(0,1)      (1,0)     (1,1)
(0,-0.01)  (10.5,-1) (2.5,-2.5)
```

The file can be found as `read_pairs1.dat`. Our task is to read this text into a nested list `pairs` such that `pairs[i]` holds the pair with index `i`, and this pair is a tuple of two `float` objects. We assume that there are no blanks inside the parentheses of a pair of numbers (we rely on a split operation which would otherwise not work).

*Solution.* To solve this programming problem, we can read in the file line by line; for each line: split the line into words (i.e., split with respect to whitespace); for each word: strip off the parentheses, split with respect to comma, and convert the resulting two words to floats. Our brief algorithm can be almost directly translated to Python code:

```
lines = open('read_pairs1.dat', 'r').readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1]  # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1);  n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)  # add 2-tuple to last row
```

This code is available in the file `read_pairs1.py`. Figure 6.3 shows a snapshot of the state of the variables in the program after having treated the first line. You should explain each line in the program to yourself, and compare your understanding with the figure.

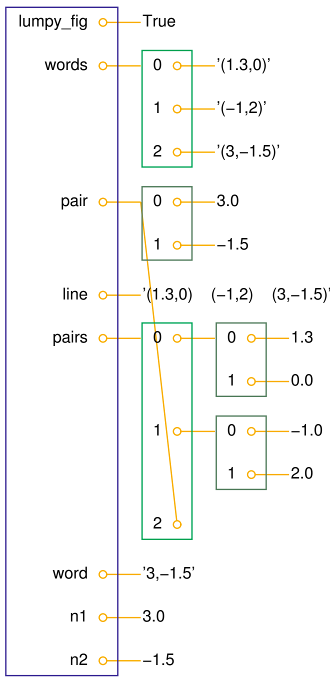



**Fig. 6.3** Illustration of the variables in the `read_pairs.py` program after the first pass in the loop over words in the first line of the data file.

The output from the program becomes

```
[(1.3, 0.0),
 (-1.0, 2.0),
 (3.0, -1.5),
 (0.0, 1.0),
```

```
    (1.0, 0.0),
    (1.0, 1.0),
    (0.0, -0.01),
    (10.5, -1.0),
    (2.5, -2.5)]
```

We remark that our solution to this programming problem relies heavily on the fact that spaces inside the parentheses are not allowed. If spaces were allowed, the simple split to obtain the pairs on a line as words would not work. What can we then do?

We can first strip off all blanks on a line, and then observe that the pairs are separated by the text `')('`. The first and last pair on a line will have an extra parenthesis that we need to remove. The rest of code is similar to the previous code and can be found in `read_pairs2.py`:

```
infile = open('read_pairs2.dat', 'r')
lines = infile.readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    line = line.strip()  # remove whitespace such as newline
    line = line.replace(' ', '')  # remove all blanks
    words = line.split(')(')
    # strip off leading/trailing parenthesis in first/last word:
    words[0] = words[0][1:]      # (-1,3  ->  -1,3
    words[-1] = words[-1][:-1]   # 8.5,9) ->  8.5,9
    for word in words:
        n1, n2 = word.split(',')
        n1 = float(n1);  n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)

infile.close()
```

The program can be tested on the file `read_pairs2.dat`:

```
   (1.3 , 0)    (-1 , 2 )    (3, -1.5)
(0 , 1)      ( 1, 0)      ( 1 , 1 )
(0,-0.01)  (10.5,-1) (2.5, -2.5)
```

A third approach is to notice that if the pairs were separated by commas,

```
(1, 3.0),    (-1, 2),    (3, -1.5)
(0, 1),      (1, 0),     (1, 1)
```

the file text is very close to the Python syntax of a list of 2-tuples. By adding enclosing brackets, plus a comma at the end of each line,

```
[(1, 3.0),    (-1, 2),    (3, -1.5),
(0, 1),      (1, 0),     (1, 1),]
```

we have a string to which we can apply `eval` to get the `pairs` list directly. Here is the code doing this (program `read_pairs3.py`):

```
infile = open('read_pairs3.dat', 'r')
listtext = '['
for line in infile:
    # add line, without newline (line[:-1]), with a trailing comma:
    listtext += line[:-1] + ', '
infile.close()
listtext = listtext + ']'
pairs = eval(listtext)
```

In general, it is a good idea to construct file formats that are as close as possible to valid Python syntax such that one can take advantage of the `eval` or `exec` functions to turn text into "live objects".

### 6.3.3 Example: Reading Coordinates

*Problem.* Suppose we have a file with coordinates $(x, y, z)$ in three-dimensional space. The file format looks as follows:

```
x=-1.345      y= 0.1112      z= 9.1928
x=-1.231      y=-0.1251      z= 1001.2
x= 0.100      y= 1.4344E+6   z=-1.0100
x= 0.200      y= 0.0012      z=-1.3423E+4
x= 1.5E+5     y=-0.7666      z= 1027
```

The goal is to read this file and create a list with `(x,y,z)` 3-tuples, and thereafter convert the nested list to a two-dimensional array with which we can compute.

Note that there is sometimes a space between the `=` signs and the following number and sometimes not. Splitting with respect to space and extracting every second word is therefore not an option. We shall present three solutions.

*Solution 1: Substring Extraction.* The file format looks very regular with the `x=`, `y=`, and `z=` texts starting in the same columns at every line. By counting characters, we realize that the `x=` text starts in column 2, the `y=` text starts in column 16, while the `z=` text starts in column 31. Introducing

```
x_start = 2
y_start = 16
z_start = 31
```

the three numbers in a `line` string are obtained as the substrings

```
x = line[x_start+2:y_start]
y = line[y_start+2:z_start]
z = line[z_start+2:]
```

The following code, found in file `file2coor_v1.py`, creates the `coor` array with shape $(n, 3)$, where $n$ is the number of $(x, y, z)$ coordinates.

```
infile = open('xyz.dat', 'r')
coor = []  # list of (x,y,z) tuples
for line in infile:
    x_start = 2
    y_start = 16
    z_start = 31
    x = line[x_start+2:y_start]
    y = line[y_start+2:z_start]
    z = line[z_start+2:]
    print 'debug: x="%s", y="%s", z="%s"' % (x,y,z)
    coor.append((float(x), float(y), float(z)))
infile.close()
```

```
from numpy import *
coor = array(coor)
print coor.shape, coor
```

The `print` statement inside the loop is always wise to include when doing string manipulations, simply because counting indices for substring limits quickly leads to errors. Running the program, the output from the loop looks like this

```
debug: x="-1.345   ", y=" 0.1112   ", z=" 9.1928
"
```

for the first line in the file. The double quotes show the exact extent of the extracted coordinates. Note that the last quote appears on the next line. This is because `line` has a newline at the end (this newline must be there to define the end of the line), and the substring `line[z_start:]` contains the newline at the of `line`. Writing `line[z_start:-1]` would leave the newline out of the $z$ coordinate. However, this has no effect in practice since we transform the substrings to `float`, and an extra newline or other blanks make no harm.

The `coor` object at the end of the program has the value

```
[[ -1.34500000e+00   1.11200000e-01   9.19280000e+00]
 [ -1.23100000e+00  -1.25100000e-01   1.00120000e+03]
 [  1.00000000e-01   1.43440000e+06  -1.01000000e+00]
 [  2.00000000e-01   1.20000000e-03  -1.34230000e+04]
 [  1.50000000e+05  -7.66600000e-01   1.02700000e+03]]
```

*Solution 2: String Search.* One problem with the solution approach above is that the program will not work if the file format is subject to a change in the column positions of `x=`, `y=`, or `z=`. Instead of hardcoding numbers for the column positions, we can use the `find` method in string objects to locate these column positions:

```
x_start = line.find('x=')
y_start = line.find('y=')
z_start = line.find('z=')
```

The rest of the code is similar to the complete program listed above, and the complete code is stored in the file `file2coor_v2.py`.

*Solution 3: String Split.* String splitting is a powerful tool, also in the present case. Let us split with respect to the equal sign. The first line in the file then gives us the words

```
['x', '-1.345   y', ' 0.1112   z', ' 9.1928']
```

We throw away the first word, and strip off the last character in the next word. The final word can be used as is. The complete program is found in the file `file2coor_v3.py` and looks like

```
infile = open('xyz.dat', 'r')
coor = []  # list of (x,y,z) tuples
for line in infile:
    words = line.split('=')
```

```
    x = float(words[1][:-1])
    y = float(words[2][:-1])
    z = float(words[3])
    coor.append((x, y, z))
infile.close()

from numpy import *
coor = array(coor)
print coor.shape, coor
```

More sophisticated examples of string operations appear in Chapter 6.4.4.

## 6.4 Reading Data from Web Pages

Python has a module `urllib` which makes it possible to read data from a web page as easily as we can read data from an ordinary file[14]. Before we do this, a few concepts from the Internet world must be touched.

### 6.4.1 About Web Pages

Web pages are viewed with a web browser. There are many competing browsers, although most people have only heard of Internet Explorer from Microsoft. Mac users may prefer Safari, while Firefox and Opera are browsers that run on different types of computers, including Windows, Linux, and Mac.

Any web page you visit is associated with an address, usually something like

```
http://www.some.where.net/some/file.html
```

This type of web address is called a URL (which stands for Uniform Resource Locator[15]). The graphics you see in a web browser, i.e., the web page you see with your eyes, is produced by a series of commands that specifies the text on the page, the images, buttons to be pressed, etc. Roughly speaking, these commands are like statements in computer programs. The commands are stored in a text file and follow rules in a language, exactly as you are used to when writing statements in a programming language.

A common language for defining web pages is HTML. A web page is then simply a text file with text containing HTML commands. Instead

[14] In principle this is true, but in practice the text in web pages tend to be much more complicated than the text in the files we have treated so far.

[15] Another term is URI (Uniform Resource Identifier), which is replacing URL in technical documentation. We stick to URL, however, in this book because Python's tools for accessing resources on the Internet have `url` as part of module and function names.

of a physical file, the web page can also be the output text from a program. In that case the URL is the name of the program file. The web browser interprets the text and the commands, and displays the information visually. Let us demonstrate this for a very simple web page shown in Figure 6.4. This page was produced by the following




**Fig. 6.4** Example of what a very simple HTML file looks like in a web browser.

text with embedded HTML commands:

```
<html>
<body bgcolor="orange">
<h1>A Very Simple HTML Page</h1> <!-- headline -->
Web pages are written in a language called
<a href="http://www.w3.org/MarkUp/Guide/">HTML</a>.
Ordinary text is written as ordinary text, but when we
need links, headlines, lists,
<ul>
<li><em>emphasized words</em>, or
<li> <b>boldface text</b>,
</ul>
we need to embed the text inside HTML tags. We can also
insert GIF or PNG images, taken from other Internet sites,
if desired.
<hr> <!-- horizontal line -->
<img src="http://www.simula.no/simula_logo.gif">
</body>
</html>
```

A typical HTML command consists of an opening and a closing *tag*. For example, emphasized text is specified by enclosing the text inside `em` (emphasize) tags:

```
<em>emphasized words</em>
```

The opening tag is enclosed in less than and greater than signs, while the closing tag has an additional forward slash before the tag name.

In the HTML file we see an opening and closing `html` tag around the whole text in the file. Similarly, there is a pair of `body` tags, where the first one also has a parameter `bgcolor` which can be used to specify a background color in the web page. Section headlines are specified by enclosing the headline text inside `h1` tags. Subsection headlines apply `h2` tags, which results in a smaller font compared with `h1` tags. Comments appear inside `<!-` and `->`. Links to other web pages are written inside

a tags, with an argument href for the link's web address. Lists apply the ul (unordered list) tag, while each item is written with just an opening tag li (list item), but no closing tag is necessary. Images are also specified with just an opening tag having name img, and the image file is given as a file name or URL of a file, enclosed in double quotes, as the src parameter.

The ultra-quick HTML course in the previous paragraphs gives a glimpse of how web pages can be constructed. One can either write the HTML file by hand in a pure text editor, or one can use programs such as Dream Weaver to help design the page graphically in a user-friendly environment, and then the program can automatically generate the right HTML syntax in files.

### 6.4.2 How to Access Web Pages in Programs

Why is it useful to know some HTML and how web pages are constructed? The reason is that the web is full of information that we can get access to through programs and use in new contexts. What we can get access to is not the visual web page you see, but the underlying HTML file. The information you see on the screen appear in text form in the HTML file, and by extracting text, we can get hold of the text's information in a program.

Given the URL as a string stored in a variable, there are two ways of accessing the HTML text in a Python program:

1. Download the HTML file and store it as a local file with a given name, say webpage.html:

```
import urllib
url = 'http://www.simula.no/research/scientific/cbc'
urllib.urlretrieve(url, filename='webpage.html')
```

2. Open the HTML file as a file-like object:

```
infile = urllib.urlopen(url)
```

This f has methods such as read, readline, and readlines.

### 6.4.3 Example: Reading Pure Text Files

HTML files are often like long and complicated programs. The information you are interested in might be buried in lots of HTML tags and ugly syntax. Extracting data from HTML files in Python programs is therefore not always an easy task. How to approach this problem is exemplified in Chapter 6.4.4. Nevertheless, the world of numerical computing can occasionally be simpler: data that we are interested in

computing with often appear in plain text files which can be accessed by URLs and viewed in web browsers. One example is temperature data from cities around the world. The temperatures in Oslo from Jan 1, 1995, until today are found in the URL

```
ftp://ftp.engr.udayton.edu/jkissock/gsod/NOOSLO.txt
```

The data reside in an ordinary text file, because the URL ends with a text file name `NOOSLO.txt`. This text file can be downloaded via the URL:

```
import urllib
url = 'ftp://ftp.engr.udayton.edu/jkissock/gsod/NOOSLO.txt'
urllib.urlretrieve(url, filename='Oslo.txt')
```

By looking at the file `Oslo.txt` in a text editor, or simply by watching the web page in a web browser, we see that the file contains four columns, with the number of the month (1-12), the date, the year, and the temperature that day (in Fahrenheit). Here is an extract of the file:

```
    1             1             1995        24.0
    1             2             1995        22.4
    1             3             1995        9.3
    1             4             1995        6.1
    1             5             1995        26.2
    1             6             1995        24.8
    1             7             1995        27.9
    1             8             1995        33.0
    1             9             1995        31.4
    1             10            1995        29.4
    1             11            1995        23.0
```

How can we load these file data into a Python program? First, we must decide on the data structure for storing the data. A nested dictionary could be handy for this purpose: `temp[year][month][date]`. That is, given the year, the number of the month, and the date as keys, the value of `temp` yields the corresponding temperature. The process of loading the file into such a dictionary is then a matter of reading the file line by line, and for each line, split the line into words, use the three first words as keys and the last as value.

A first attempt to load the file data into the `temps` dictionary could be like

```
infile = open('Oslo.txt', 'r')
temps = {}
for line in infile:
    month, date, year, temperature = line.split()
    temps[year][month][date] = temperature
```

However, running these lines gives a `KeyError: '1995'`. One can normally just assign new keys to a dictionary, but this is a nested dictionary, and each level must be initialized, not just the first level (which is initialized in the line before the `for` loop). Recall that for a period of 15 years, `temps` will consist of 15 dictionaries, and each of these contains 12 dictionaries for the months. This fact complicates the code:

We must test if a year or month key is present, and if not, an empty dictionary must be assigned. We must also transform the strings returned by `line.split()` into `int` and `float` objects (month, date, and year are integers while the temperature is a real number). The complete program is listed below and available in the file `webtemps.py`.

```
import urllib
url = 'ftp://ftp.engr.udayton.edu/jkissock/gsod/NOOSLO.txt'
urllib.urlretrieve(url, filename='Oslo.txt')

infile = open('Oslo.txt', 'r')
temps = {}
for line in infile:
    month, date, year, temperature = line.split()
    month = int(month)
    date = int(date)
    year = int(year)
    temperature = float(temperature)
    if not year in temps:
        temps[year] = {}
    if not month in temps[year]:
        temps[year][month] = {}
    temps[year][month][date] = temperature
infile.close()

# pick a day to verify that temps is correct:
year = 2003; month = 3; date = 31
T = temps[year][month][date]
print '%d.%d.%d: %.1f' % (year, month, date, T)
```

Running this code results in

```
2003.3.31: 38.5
```

We can view the `Oslo.txt` file and realize that the printed temperature is correct.

### 6.4.4 Example: Extracting Data from an HTML Page

The weather forcast on Yahoo! contains a lot of graphics and advertisements along with weather data. Suppose you want to quickly see today's weather and temperature in a city. Can we make a program that finds this information in the web page? The answer is yes if we can download the page with `urllib.urlretrieve` (or open the web page as a file with `urllib.urlopen`) and if we know some string operations to help us search for some specific text.

The Yahoo! page for the weather in Oslo is (at the time of this writing) found at

```
http://weather.yahoo.com/forecast/NOXX0029_c.html
```

All the text and images on this weather page are defined in the file associated with this address. First we download the file,

```
import urllib
w = 'http://weather.yahoo.com/forecast/NOXX0029_c.html'
urllib.urlretrieve(url=w, filename='weather.html')
```

Since the weather and temperature data are known to reside somewhere inside this file, we take a look at the downloaded file, named `weather.html`, in a pure text editor. What we see, is quite complicated text like

```
<!-- BEGIN TOP SEARCH -->
<div id="ynneck">
 <div id="searchbartop" class="searchbar">
  <form action="http://news.search.yahoo.com/news/search" method="get">
  <strong>Search:</strong>
  <input type="text" name="p" size="30">
  <!-- Include search dropdown box -->

  <select name=c>
  <option value="">All News &amp; Blogs</option>
  <option value=yahoo_news>Yahoo! News Only</option>
  <option value=news_photos>News Photos</option>
  <option value=av>Video/Audio</option>
```

The file is written in the HTML format, using a lot of different tag names. Most of the text is uninteresting to us, but we see in the web browser that the data we are looking for appear under a text "Current conditions ...". Searching for "Current conditions" in the `weather.html` file brings us down to the middle of the file. Here, some interesting text is embedded in some surrounding, uninteresting text:

```
<div class="forecast-module">
<em>Current conditions as of 2:19 pm CET</em>
<h3>Mostly Cloudy</h3>


<dl>
<dt>Feels Like:</dt>
<dd>2&deg;</dd>
<dt>Barometer:</dt>

<dd>--</dd>
<dt>Humidity:</dt>
<dd>53%</dd>
<dt>Visibility:</dt>
<dd>
9.99 km</dd>

<dt>Dewpoint:</dt>
<dd>-3&deg;</dd>

<dt>Wind:</dt>
<dd>N 24 kph</dd>

<dt>Sunrise:</dt>
<dd>6:19 am</dd>
<dt>Sunset:</dt>
<dd>
6:29 pm</dd>
</dd>
</dl>

<div id="forecast-temperature">
<h3>6&deg;</h3>
<p>High: 4&deg; Low: -2&deg;</p>
```

We are interested in two pieces of text:

1. After the line containing the text “Current conditions”, we have a line stating today’s weather:

```
<em>Current conditions as of 2:19 pm CET</em>
<h3>Mostly Cloudy</h3>
```

2. After the line containing “forecast-temperature”, we have a line with today’s temperature:

```
<div id="forecast-temperature">
<h3>6&deg;</h3>
```

The text `°` is a special HTML command for the degrees symbol. We are not interested in this symbol, but the number prior to it.

We can extract the weather description and the temperature by running through all lines in the file, check if a line contains either “Current conditions” or “forecast-temperature”, and in both cases extract information from the next line. Since we need to go through all lines and look at the line after the current one, it is a good strategy to load all lines into a list and traverse the list with a `for` loop over list indices, because when we are at a line with index `i`, we can easily look at the next one with index `i+1`:

```
lines = infile.readlines()
for i in range(len(lines)):
    line = lines[i]  # short form
    if 'Current conditions' in line:
        weather = lines[i+1][4:-6]
    if 'forecast-temperature' in line:
        temperature = float(lines[i+1][4:].split('&')[0])
        break  # everything is found, jump out of loop
```

The lines computing `weather` and `temperature` probably need some comments. Looking at a line containing the text which `weather` should be set equal to,

```
<h3>Mostly Cloudy</h3>
```

we guess that in the general case we are intersted in the text between `<h3>` and `</h3>`. The text in the middle can be extracted as a substring. Since we do not know the length of the weather description, we count from the right when creating the substring. The substring starts from index 4 and should go to, but not include, index -6 (you might think it would be index -5, but there is an invisible newline character at the end of the `line` string which we must also count). A small test in an interactive session can be used to control that we are doing the right thing:

```
>>> s = "<h3>Mostly Cloudy</h3>\n"
>>> s[4:-6]
'Mostly Cloudy'
```

The extraction of the temperature is more involved. The actual line looks like

```
<h3>6&deg;</h3>
```

We want to extract the number that comes after `<h3>` and before the ampersand sign. One method is to strip off the leading `<h3>` text by extracting the substring `lines[i+1][4:]`. Then we can split this substring with respect to the ampersand character. The first “word” from this split is the temperature, but the type is a string, so we need to convert it to `float` to be ready for calculations with the temperature number. All these steps can be done separately to better explain the individual tasks:

```
next_line = lines[i+1]
substring = next_line[4:]
words = substring.split('&')
temperature = words[0]
temperature = float(temperature)
```

The experienced programmer often prefers to condense the code and combine the five statements into one:

```
temperature = float(lines[i+1][4:].split('&')[0])
```

Note that when we have found the temperature, there is no need to examine more lines in the file. We therefore execute a `break` statement, which forces the program control to jump out of the loop.

Finally, we wrap the code extracting the weather and temperature conditions in a function:

```
def get_data(url):
    urllib.urlretrieve(url=url, filename='weather.html')
    infile = open('weather.html')
    lines = infile.readlines()
    ...
    infile.close()
    return weather, temperature
```

We might visit the Yahoo! weather pages and pick out a collection of cities and corresponding URLs, to be stored in a dictionary:

```
cities = {
    'Sandefjord':
    'http://weather.yahoo.com/forecast/NOXX0032_c.html',
    'Oslo':
    'http://weather.yahoo.com/forecast/NOXX0029_c.html',
    'Gothenburg':
    'http://weather.yahoo.com/forecast/SWXX0007_c.html',
    'Copenhagen':
    'http://weather.yahoo.com/forecast/DAXX0009_c.html',
    }
```

A simple loop over this dictionary and a call to `get_data` gives a quick look at the weather conditions and temperatures in several cities:

```
for city in cities:
    weather, temperature = get_data(cities[city])
    print city, weather, temperature
```

The complete code is found in the file `weather.py`. A sample run on a winter day gave this result:

```
Oslo Fair 3.0
Copenhagen Partly Cloudy 5.0
Sandefjord Fair 4.0
Gothenburg Fair 0.0
```

The techniques described above are useful when you need to extract data from the web and process the data, either for presentation in a compact format as above or for further calculations. Turning web page information into compact text can also be useful for constructing SMS messages to be sent to mobile phones.

*Remark.* Interpretation of text in files is based on string operations in this book. A much more powerful and often more convenient approach is to apply so-called *regular expressions*. An introduction to regular expressions of relevance to scientific computations is given in the book [5]. We strongly recommend to learn about regular expressions if you end up interpreting a lot of HTML text in web pages. An even more powerful technique to extract information from web pages is to use an HTML parser, which comes with standard Python. This technique requires more programming compared to using string operations or regular expressions, but can handle cases that are impossible or difficult to address with the two other techniques. Googling for "HTML parsing Python" gives a lot of links to what HTML parsing is about and how it is done.

## 6.5 Writing Data to File

Writing data to file is easy. There is basically one function to pay attention to: `outfile.write(s)`, which writes a string `s` to a file handled by the file object `outfile`. Unlike `print`, `outfile.write(s)` does not append a newline character to the written string. It will therefore often be necessary to add a newline character,

```
outfile.write(s + '\n')
```

if the string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character. File writing is then a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`.

An alternative syntax to `outfile.write(s)` is `print ■ f, s`. This `print` statement adds a newline character, as usual.

Writing to a file demands the file object `f` to be opened for writing:

```
# write to new file, or overwrite file:
outfile = open(filename, 'w')

# append to the end of an existing file:
outfile = open(filename, 'a')
```

### 6.5.1 Example: Writing a Table to File

*Problem.* As a worked example of file writing, we shall write out a nested list with tabular data to file. A sample list may take look as

```
[[ 0.75,        0.29619813, -0.29619813, -0.75      ],
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
 [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
```

*Solution.* We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak. The code resides in the file `write1.py`:

```
data = [[ 0.75,        0.29619813, -0.29619813, -0.75      ],
        [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
        [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
        [-0.75,       -0.29619813,  0.29619813,  0.75      ]]

outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```

The resulting data file becomes

```
    0.75000000    0.29619813   -0.29619813   -0.75000000
    0.29619813    0.11697778   -0.11697778   -0.29619813
   -0.29619813   -0.11697778    0.11697778    0.29619813
   -0.75000000   -0.29619813    0.29619813    0.75000000
```

An extension of this program consists in adding column and row headings:

```
           column  1     column  2     column  3     column  4
row  1    0.75000000    0.29619813   -0.29619813   -0.75000000
row  2    0.29619813    0.11697778   -0.11697778   -0.29619813
row  3   -0.29619813   -0.11697778    0.11697778    0.29619813
row  4   -0.75000000   -0.29619813    0.29619813    0.75000000
```

To obtain this end result, we need to the add some statements to the program `write1.py`. For the column headings we need to know the number of columns, i.e., the length of the rows, and loop from 1 to this length:

```
ncolumns = len(data[0])
outfile.write('          ')
for i in range(1, ncolumns+1):
    outfile.write('%10s    ' % ('column %2d' % i))
outfile.write('\n')
```

Note the use of a nested printf construction: The text we want to insert is itself a printf string. We could also have written the text as `'column ' + str(i)`, but then the length of the resulting string would depend on the number of digits in `i`. It is recommended to always use printf constructions for a tabular output format, because this gives automatic padding of blanks so that the width of the output strings remain the same. As always, the tuning of the widths is done in a trial-and-error process.

To add the row headings, we need a counter over the row numbers:

```
row_counter = 1
for row in data:
    outfile.write('row %2d' % row_counter)
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
    row_counter += 1
```

The complete code is found in the file `write2.py`. We could, alternatively, iterate over the indices in the list:

```
for i in range(len(data)):
    outfile.write('row %2d' % (i+1))
    for j in range(len(data[i])):
        outfile.write('%14.8f' % data[i][j])
    outfile.write('\n')
```

### 6.5.2 Standard Input and Output as File Objects

Reading user input from the keyboard applies the function `raw_input` as explained in Chapter 3.1. The keyboard is a medium that the computer in fact treats as a file, referred to as *standard input*.

The `print` command prints text in the terminal window. This medium is also viewed as a file from the computer's point of view and called *standard output*. All general-purpose programming languages allow reading from standard input and writing to standard output. This reading and writing can be done with two types of tools, either file-like objects or special tools like `raw_input` (see Chapter 3.1.1) and `print` in Python. We will here describe the file-line objects: `sys.stdin` for standard input and `sys.stdout` for standard output. These objects behave as file objects, except that they do not need to be opened or closed. The statement

```
s = raw_input('Give s:')
```

is equivalent to

```
print 'Give s: ',
s = sys.stdin.readline()
```

Recall that the trailing comma in the `print` statement avoids the new-line that `print` by default adds to the output string. Similarly,

```
s = eval(raw_input('Give s:'))
```

is equivalent to

```
print 'Give s: ',
s = eval(sys.stdin.readline())
```

For output to the terminal window, the statement

```
print s
```

is equivalent to

```
sys.stdout.write(s + '\n')
```

Why it is handy to have access to standard input and output as file objects can be illustrated by an example. Suppose you have a function that reads data from a file object `infile` and writes data to a file object `outfile`. A sample function may take the form

```
def x2f(infile, outfile, f):
    for line in infile:
        x = float(line)
        y = f(x)
        outfile.write('%g\n' % y)
```

This function works with all types of files, including web pages as `infile` (see Chapter 6.4). With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, the `x2f` function also works with standard input and/or standard output. Without `sys.stdin` and `sys.stdout`, we would need different code, employing `raw_input` and `print`, to deal with standard input and output. Now we can write a single function that deals with all file media in a unified way.

There is also something called *standard error*. Usually this is the terminal window, just as standard output, but programs can distinguish between writing ordinary output to standard output and error messages to standard error, and these output media can be redirected to, e.g., files such that one can separate error messages from ordinary output. In Python, standard error is the file-like object `sys.stderr`. A typical application of `sys.stderr` is to report errors:

```
if x < 0:
    sys.stderr.write('Illegal value of x'); sys.exit(1)
```

This message to `sys.stderr` is an alternative to `print` or raising an exception.

*Redirecting Standard Input, Output, and Error.* Standard output from a program `prog` can be redirected to a file `out` using the greater than sign[16]:

Terminal
```
Unix/DOS> prog > output
```

That is, the program writes to `output` instead of the terminal window. Standard error can be redirected by

Terminal
```
Unix/DOS> prog &> output
```

When the program reads from standard input, we can equally well redirect standard input to a file, say with name `raw_input`, such that the program reads from this file rather than from the keyboard:

Terminal
```
Unix/DOS> prog < input
```

Combinations are also possible:

Terminal
```
Unix/DOS> prog < input > output
```

*Note.* The redirection of standard output, input, and error does not work for programs run inside IPython, only when run directly in the operating system in a terminal window.

Inside a Python program we can also let standard input, output, and error work with ordinary files instead. Here is the technique:

```
sys_stdout_orig = sys.stdout
sys.stdout = open('output', 'w')
sys_stdin_orig = sys.stdin
sys.stdin = open('input', 'r')
```

Now, any `print` statement will write to the `output` file, and any `raw_input` call will read from the `input` file. (Without storing the originalsys.stdout and `sys.stdin` objects in new variables, these objects would get lost in the redefinition above and we would never be able to reach the common standard input and output in the program.)

### 6.5.3 Reading and Writing Spreadsheet Files

From school you are probably used to spreadsheet programs such as Microsoft Excel or OpenOffice. This type of program is used to represent a table of numbers and text. Each table entry is known as a

[16] `prog` can be any program, including a Python program run as, e.g., `python myprog.py`.

*cell*, and one can easily perform calculations with cells that contain numbers. The application of spreadsheet programs for mathematical computations and graphics is steadily growing.

Also Python may be used to do spreadsheet-type calculations on tabular data. The advantage of using Python is that you can easily extend the calculations far beyond what a spreadsheet program can do. However, even if you can view Python as a substitute for a spreadsheet program, it may be beneficial to combine the two. Suppose you have some data in a spreadsheet. How can you read these data into a Python program, perform calculations on the data, and thereafter read the data back to the spreadsheet program? This is exactly what we will explain below through an example. With this example, you should understand how easy it is to combine Excel or OpenOffice with your own Python programs.

The table of data in a spreadsheet can be saved in so-called CSV files, where CSV stands for *comma separated values*. The CSV file format is very simple: each row in the spreadsheet table is a line in the file, and each cell in the row is separated by a comma or some other specified separation character. CSV files can easily be read into Python programs, and the table of cell data can be stored in a nested list (table, cf. Chapter 2.1.7), which can be processed as we desire. The modified table of cell data can be written back to a CSV file and read into the spreadsheet program for further processing.

Figure 6.5 shows a simple spreadsheet in the OpenOffice program. The table contains $4 \times 4$ cells, where the first row contains column headings and the first column contains row headings. The remaining $3 \times 3$ subtable contains numbers that we may compute with. Let us

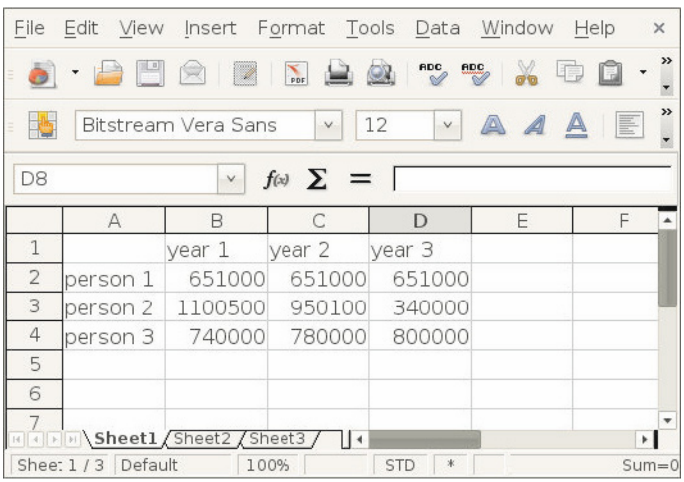



**Fig. 6.5** A simple spreadsheet in OpenOffice.

save this spreadsheet to a file in the CSV format. The complete file will typically look as follows:

```
,"year 1","year 2","year 3"
"person 1",651000,651000,651000
"person 2",1100500,950100,340000
"person 3",740000,780000,800000
```

*Reading CSV Files.* Our goal is to write a Python code for loading the spreadsheet data into a table. The table is technically a nested list, where each list element is a row of the table, and each row is a list of the table's column values. CSV files can be read, row by row, using the `csv` module from Python's standard library. The recipe goes like this, if the data reside in the CSV file `budget.csv`:

```
infile = open('budget.csv', 'r')
import csv
table = []
for row in csv.reader(infile):
    table.append(row)
infile.close()
```

The `row` variable is a list of column values that are read from the file by the `csv` module. The three lines computing `table` can be condensed to one using a list comprehension:

```
table = [row for row in csv.reader(infile)]
```

We can easily print `table`,

```
import pprint
pprint.pprint(table)
```

to see what the spreadsheet looks like when it is represented as a nested list in a Python program:

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', '651000', '651000', '651000'],
 ['person 2', '1100500', '950100', '340000'],
 ['person 3', '740000', '780000', '800000']]
```

Observe now that all entries are surrounded by quotes, which means that all entries are string (`str`) objects. This is a general rule: the `csv` module reads all cells into string objects. To compute with the numbers, we need to transform the string objects to `float` objects. The transformation should not be applied to the first row and first column, since the cells here hold text. The transformation from strings to numbers therefore applies to the indices `r` and `c` in `table` (`table[r][c]`), such that the row counter `r` goes from 1 to `len(table)-1`, and the column counter `c` goes from 1 to `len(table[0])-1` (`len(table[0])` is the length of the first row, assuming the lengths of all rows are equal to the length of the first row). The relevant Python code for this transformation task becomes

```
for r in range(1,len(table)):
   for c in range(1, len(table[0])):
       table[r][c] = float(table[r][c])
```

A `pprint.pprint(table)` statement after this transformation yields

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', 651000.0, 651000.0, 651000.0],
 ['person 2', 1100500.0, 950100.0, 340000.0],
 ['person 3', 740000.0, 780000.0, 800000.0]]
```

The numbers now have a decimal and no quotes, indicating that the numbers are `float` objects and hence ready for mathematical calulations.

*Processing Data.* Let us perform a very simple calculation with `table`, namely adding a final row with the sum of the numbers in the columns:

```
row = [0.0]*len(table[0])
row[0] = 'sum'
for c in range(1, len(row)):
    s = 0
    for r in range(1, len(table)):
        s += table[r][c]
    row[c] = s
```

As seen, we first create a list `row` consisting of zeros. Then we insert a text in the first column, before we invoke a loop over the numbers in the table and compute the sum of each column. The `table` list now represents a spreadsheet with four columns and five rows:

```
[['', 'year 1', 'year 2', 'year 3'],
 ['person 1', 651000.0, 651000.0, 651000.0],
 ['person 2', 1100500.0, 950100.0, 340000.0],
 ['person 3', 740000.0, 780000.0, 800000.0],
 ['sum', 2491500.0, 2381100.0, 1791000.0]]
```

*Writing CSV Files.* Our final task is to write the modified `table` list back to a CSV file so that the data can be loaded in a spreadsheet program. The write task is done by the code segment

```
outfile = open('budget2.csv', 'w')
writer = csv.writer(outfile)
for row in table:
    writer.writerow(row)
outfile.close()
```

The `budget2.csv` looks like this:

```
,year 1,year 2,year 3
person 1,651000.0,651000.0,651000.0
person 2,1100500.0,950100.0,340000.0
person 3,740000.0,780000.0,800000.0
sum,2491500.0,2381100.0,1791000.0
```

The final step is to read `budget2.csv` into a spreadseet. The result is displayed in Figure 6.6 (in OpenOffice one must specify in the "open" dialog that the spreadsheet data are separated by commas, i.e., that the file is in CSV format).

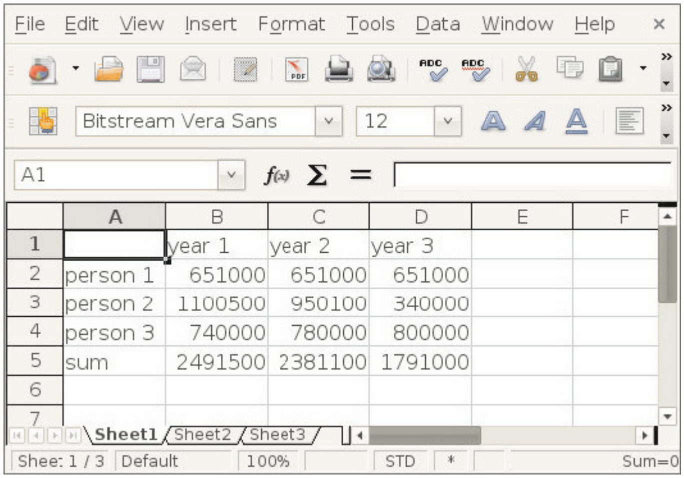



**Fig. 6.6** A spreadsheet processed in a Python program and loaded back into OpenOffice.

The complete program reading the `budget.csv` file, processing its data, and writing the `budget2.csv` file can be found in `rw_csv.py`. With this example at hand, you should be in a good position to combine spreadsheet programs with your own Python programs.

*Remark.* You may wonder why we used the `csv` module to read and write CSV files when such files have comma separated values which we can extract by splitting lines with respect to the comma (in Chapter 6.2.6 use used this technique to read a CSV file):

```
infile = open('budget.csv', 'r')
for line in infile:
    row = line.split(',')
```

This works well for the present `budget.csv` file, but the technique breaks down when a text in a cell contains a comma, for instance `"Aug 8, 2007"`. The `line.split(',')` will split this cell text, while the `csv.reader` functionality is smart enough to avoid splitting text cells with a comma.

*Representing Number Cells with Numerical Python Arrays.* Instead of putting the whole spreadsheet into a single nested list, we can make a Python data structure more tailored to the data at hand. What we have are two headers (for rows and columns, respectively) and a subtable of numbers. The headers can be represented as lists of strings, while the subtable could be a two-dimensional Numerical Python array. The latter makes it easier to implement various mathematical operations on the numbers. A dictionary can hold all the three items: two header lists and one array. The relevant code for reading, processing, and writing the data is shown below and can be found in the file `rw_csv_numpy.py`:

```
infile = open('budget.csv', 'r')
import csv
table = [row for row in csv.reader(infile)]
infile.close()

# convert subtable of numbers (string to float):
subtable = [[float(c) for c in row[1:]] for row in table[1:]]

data = {'column headings': table[0][1:],
        'row headings': [row[0] for row in table[1:]],
        'array': array(subtable)}

# add a new row with sums:
data['row headings'].append('sum')
a = data['array']   # short form
data['column sum'] = [sum(a[:,c]) for c in range(a.shape[1])]

outfile = open('budget2.csv', 'w')
writer = csv.writer(outfile)
# turn data dictionary into a nested list first (for easy writing):
table = a.tolist()   # transform array to nested list
table.append(data['column sum'])
table.insert(0, data['column headings'])
# extend table with row headings (a new column):
table = [table[r].insert(0, data['row headings'][r]) \
         for r in range(len(table))]
for row in table:
    writer.writerow(row)
outfile.close()
```

The code makes heavy use of list comprehensions, and the transformation between a nested list, for file reading and writing, and the `data` dictionary, for representing the data in the Python program, is non-trivial. If you manage to understand every line in this program, you have digested a lot of topics in Python programming!

## 6.6 Summary

### 6.6.1 Chapter Topics

*File Operations.* This chapter has been concerned with file reading and file writing. First a file must be openend, either for reading, writing, or appending:

```
infile  = open(filename, 'r')   # read
outfile = open(filename, 'w')   # write
outfile = open(filename, 'a')   # append
```

There are four basic reading commands:

```
line    = infile.readline()   # read the next line
filestr = infile.read()       # read rest of file into string
lines   = infile.readlines()  # read rest of file into list
for line in infile:           # read rest of file line by line
```

File writing is usually about repeatedly using the command

```
outfile.write(s)
```

where `s` is a string. Contrary to `print s`, no newline is added to `s` in `outfile.write(s)`.

When the reading and writing is finished,

```
somefile.close()
```

should be called, where `somefile` is the file object.

*Downloading Internet Files.* Internet files can be downloaded if we know their URL:

```
import urllib
url = 'http://www.some.where.net/path/thing.html'
urllib.urlretrieve(url, filename='thing.html')
```

The downloaded information is put in the local file `thing.html` in the current working folder. Alternatively, we can open the URL as a file object:

```
webpage = urllib.urlopen(url)
```

HTML files are often messy to interpret by string operations.

**Table 6.1** Summary of important functionality for dictionary objects.

| | |
|---|---|
| `a = {}` | initialize an empty dictionary |
| `a = {'point': [0,0.1], 'value': 7}` | initialize a dictionary |
| `a = dict(point=[2,7], value=3)` | initialize a dictionary w/string keys |
| `a['hide'] = True` | add new key-value pair to a dictionary |
| `a['point']` | get value corresponding to key `point` |
| `'value' in a` | `True` if `value` is a key in `a` |
| `del a['point']` | delete a key-value pair from `a` |
| `a.keys()` | list of keys |
| `a.values()` | list of values |
| `len(a)` | number of key-value pairs in `a` |
| `for key in a:` | loop over keys in unknown order |
| `for key in sorted(a):` | loop over keys in alphabetic order |
| `isinstance(a, dict)` | is `True` if `a` is a dictionary |

*Dictionaries.* Array or list-like objects with text or other (fixed-valued) Python objects as indices are called dictionaries. They are very useful for storing general collections of objects in a single data structure. Table 6.1 displays some of the most important dictionary operations.

*Strings.* Some of the most useful functionalities in a string object `s` are listed below.

- Split the string into substrings separated by `delimiter`:

```
words = s.split(delimiter)
```

- Join elements in a list of strings:

```
string = delimiter.join(words[i:j])
```

- Extract substring:

```
substring = s[2:n-4]
```

- Substitute a substring by new a string:

```
modified_string = s.replace(sub, new)
```

- Search for the start (first index) of some text:

```
index = s.find(text)
if index == -1:
    print 'Could not find "%s" in "%s" (text, s)
else:
    substring = s[index:]  # strip off chars before text
```

- Check if a string contains whitespace only:

```
if s.isspace():
    ...
```

### 6.6.2 Summarizing Example: A File Database

*Problem.* We have a file containing information about the courses that students have taken. The file format consists of blocks with student data, where each block starts with the student's name (`Name:`), followed by the courses that the student has taken. Each course line starts with the name of the course, then comes the semester when the exam was taken, then the size of the course in terms of credit points, and finally the grade is listed (letters `A` to `F`). Here is an example of a file with three student entries:

```
Name: John Doe
Astronomy                          2003 fall 10 A
Introductory Physics               2003 fall 10 C
Calculus I                         2003 fall 10 A
Calculus II                        2004 spring 10 B
Linear Algebra                     2004 spring 10 C
Quantum Mechanics I                2004 fall 10 A
Quantum Mechanics II               2005 spring 10 A
Numerical Linear Algebra           2004 fall 5 E
Numerical Methods                  2004 spring 20 C

Name: Jan Modaal
Calculus I                         2005 fall 10 A
Calculus II                        2006 spring 10 A
Introductory C++ Programming       2005 fall 15 D
Introductory Python Programming    2006 spring 5 A
Astronomy                          2005 fall 10 A
```

```
Basic Philosophy                  2005 fall 10 F

Name: Kari Nordmann
Introductory Python Programming   2006 spring 5 A
Astronomy                         2005 fall 10 D
```

Our problem consists of reading this file into a dictionary `data` with the student name as key and a list of courses as value. Each element in the list of courses is a dictionary holding the course name, the semester, the credit points, and the grade. A value in the `data` dictionary may look as

```
'Kari Nordmann': [{'credit': 5,
                   'grade': 'A',
                   'semester': '2006 spring',
                   'title': 'Introductory Python Programming'},
                  {'credit': 10,
                   'grade': 'D',
                   'semester': '2005 fall',
                   'title': 'Astronomy'}],
```

Having the `data` dictionary, the next task is to print out the average grade of each student.

*Solution.* We divide the problem into two major tasks: loading the file data into the `data` dictionary, and computing the average grades. These two tasks are naturally placed in two functions.

We need to have a strategy for reading the file and interpreting the contents. It will be natural to read the file line by line, and for each line check if this is a line containing a new student's name, a course information line, or a blank line. In the latter case we jump to the next pass in the loop. When a new student name is encountered, we initialize a new entry in the `data` dictionary to an empty list. In the case of a line about a course, we must interpret the contents on that line, which we postpone a bit.

We can now sketch the algorithm described above in terms of some unfinished Python code, just to get the overview:

```
def load(studentfile):
    infile = open(studentfile, 'r')
    data = {}
    for line in infile:
        i = line.find('Name:')
        if i != -1:
            # line contains 'Name:', extract the name
            ...
        elif line.isspace():     # blank line?
            continue             # go to next loop iteration
        else:
            # this must be a course line
            # interpret the line
            ...
    infile.close()
    return data
```

If we find `'Name:'` as a substring in `line`, we must extract the name. This can be done by the substring `line[i+5:]`. Alternatively, we can split the line with respect to colon and strip off the first word:

```
words = line.split(':')
name = ' '.join(words[1:])
```

We have chosen the former strategy of extracting the name as a substring in the final program.

Each course line is naturally split into words for extracting information:

```
words = line.split()
```

The name of the course consists of a number of words, but we do not know how many. Nevertheless, we know that the final words contain the semester, the credit points, and the grade. We can hence count from the right and extract information, and when we are finished with the semester information, the rest of the `words` list holds the words in the name of the course. The code goes as follows:

```
grade = words[-1]
credit = int(words[-2])
semester = ' '.join(words[-4:-2])
course_name = ' '.join(words[:-4])
data[name].append({'title':    course_name,
                   'semester': semester,
                   'credit':   credit,
                   'grade':    grade})
```

This code is a good example of the usefulness of split and join operations when extracting information from a text.

Now to the second task of computing the average grade. Since the grades are letters we cannot compute with them. A natural way to proceed is to convert the letters to numbers, compute the average number, and then convert that number back to a letter. Conversion between letters and numbers is easily represented by a dictionary:

```
grade2number = {'A': 5, 'B': 4, 'C': 3, 'D': 2, 'E': 1, 'F': 0}
```

To convert from numbers to grades, we construct the "inverse" dictionary:

```
number2grade = {}
for grade in grade2number:
    number2grade[grade2number[grade]] = grade
```

In the computation of the average grade we should use a weighted sum such that larger courses count more than smaller courses. The weighted mean value of a set of numbers $r_i$ with weights $w_i$, $i = 0, \ldots, n-1$, is given by

$$\frac{\sum_{i=0}^{n-1} w_i r_i}{\sum_{i=0}^{n-1} w_i}\,.$$

This weighted mean value must then be rounded to the nearest integer, which can be used as key in `number2grade` to find the corresponding grade expressed as a letter. The weight $w_i$ is naturally taken as the number of credit points in the course with grade $r_i$. The whole process is performed by the following function:

```
def average_grade(data, name):
    sum = 0; weights = 0
    for course in data[name]:
        weight = course['credit']
        grade  = course['grade']
        sum += grade2number[grade]*weight
        weights += weight
    avg = sum/float(weights)
    return number2grade[round(avg)]
```

The complete code is found in the file `students.py`. Running this program gives the following output of the average grades:

```
John Doe: B
Kari Nordmann: C
Jan Modaal: C
```

One feature of the `students.py` code is that the output of the names are sorted after the last name. How can we accomplish that? A straight `for name in data` loop will visit the keys in an unknown (random) order. To visit the keys in alphabetic order, we must use

```
for name in sorted(data):
```

This default sort will sort with respect to the first character in the `name` strings. We want a sort according to the last part of the name. A tailored sort function can then be written (see Exercise 2.44 for an introduction to tailored sort functions). In this function we extract the last word in the names and compare them:

```
def sort_names(name1, name2):
    last_name1 = name1.split()[-1]
    last_name2 = name2.split()[-2]
    if last_name1 < last_name2:
        return -1
    elif last_name1 > last_name2:
        return 1
    else:
        return 0
```

We can now pass on `sort_names` to the `sorted` function to get a sequence that is sorted with respect to the last word in the students' names:

```
for name in sorted(data, sort_names):
    print '%s: %s' % (name, average_grade(data, name))
```

## 6.7 Exercises

**Exercise 6.1.** *Read a two-column data file.*

The file `src/files/xy.dat` contains two columns of numbers, corresponding to $x$ and $y$ coordinates on a curve. The start of the file looks as this:

```
-1.0000     -0.0000
-0.9933     -0.0087
-0.9867     -0.0179
-0.9800     -0.0274
-0.9733     -0.0374
```

Make a program that reads the first column into a list `x` and the second column into a list `y`. Then convert the lists to arrays, and plot the curve. Print out the maximum and minimum $y$ coordinates. (Hint: Read the file line by line, split each line into words, convert to `float`, and append to `x` and `y`.) Name of program file: `read_2columns.py` ◇

**Exercise 6.2.** *Read a data file.*

The files `density_of_water.dat` and `density_of_air.dat` files in the folder `src/files` contain data about the density of water and air (resp.) for different temperatures. The data files have some comment lines starting with `#` and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding density in the second column. The goal of this exercise is to read the density data and plot them. Let the program take the name of the data file as command-line argument, load the density data into NumPy arrays, and plot the data using circles for the data points. Demonstrate that the program can read both files. Name of program file: `read_density_data.py` ◇

**Exercise 6.3.** *Simplify the implementation of Exer. 6.1.*

Files with data in a tabular fashion are very common and so is the operation of the reading the data into arrays. Therefore, the `scitools.filetable` module offers easy-to-use functions for loading data files with columns of numbers into NumPy arrays. First read about `scitools.filetable` using `pydoc` in a terminal window (cf. page 98). Then solve Exercise 6.1 using appropriate functions from the `scitools.filetable` module. Name of program file: `read_2columns_filetable.py`. ◇

**Exercise 6.4.** *Fit a polynomial to data.*

The purpose of this exercise is to find a simple mathematical formula for the how the density of water or air depends on the temperature. First, load the density data from file as explained in Exercises 6.2 or 6.3. Then we want to experiment with NumPy utilities that can find a polynomial that approximate the density curve.

NumPy has a function `polyfit(x, y, deg)` for finding a "best fit" of a polynomial of degree `deg` to a set of data points given by the array

arguments `x` and `y`. The `polyfit` function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term. For example, given points in `x` and `y`, `polyfit(x, y, 1)` returns the coefficients `a, b` in a polynomial `a*x + b` that fits the data in the best way[17].

NumPy also has a utility `poly1d` which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print p  # prints the polynomial expression
y_fitted = p(x)
plot(x, y, 'r-', x, y_fitted, 'b-',
     legend=('data', 'fitted polynomial of degree %d' % deg'))
```

For the density–temperature relationship we want to plot the data from file and two polynomial approximations, corresponding to a 1st and 2nd degree polynomial. From a visual inspection of the plot, suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature. Make three separate plots of the Name of program file: `fit_density_data.py` ◇

**Exercise 6.5.** *Read acceleration data and find velocities.*

A file `src/files/acc.dat` contains measurements $a_0, a_1, \ldots, a_{n-1}$ of the acceleration of an object moving along a straight line. The measurement $a_k$ is taken at time point $t_k = k\Delta t$, where $\Delta t$ is the time spacing between the measurements. The purpose of the exercise is to load the acceleration data into a program and compute the velocity $v(t)$ of the object at some time $t$.

In general, the acceleration $a(t)$ is related to the velocity $v(t)$ through $v'(t) = a(t)$. This means that

$$v(t) = v(0) + \int_0^t a(\tau)d\tau\,. \tag{6.1}$$

If $a(t)$ is only known at some discrete, equally spaced points in time, $a_0, \ldots, a_{n-1}$ (which is the case in this exercise), we must compute the integral (6.1) in numerically, for example by the Trapezoidal rule:

$$v(t_k) \approx \Delta t \left(\frac{1}{2}a_0 + \frac{1}{2}a_k + \sum_{i=1}^{k-1} a_i\right), \quad 1 \le k \le n-1\,. \tag{6.2}$$

[17] More precisely, a line $y = ax+b$ is a "best fit" to the data points $(x_i, y_i)$, $i = 0, \ldots, n-1$ if $a$ and $b$ are chosen to make the sum of squared errors $R = \sum_{j=0}^{n-1}(y_j - (ax_j + b))^2$ as small as possible. This approach is known as *least squares approximation* to data and proves to be extremely useful throughout science and technology.

We assume $v(0) = 0$ so that also $v_0 = 0$.

Read the values $a_0, \ldots, a_{n-1}$ from file into an array, plot the acceleration versus time, and use (6.2) to compute one $v(t_k)$ value, where $\Delta t$ and $k \geq 1$ are specified on the command line. Name of program file: `acc2vel_v1.py`. ◇

**Exercise 6.6.** *Read acceleration data and plot velocities.*

The task in this exercise is the same as in Exercise 6.5, except that we now want to compute $v(t_k)$ for all time points $t_k = k\Delta t$ and plot the velocity versus time. Repeated use of (6.2) for all $k$ values is very inefficient. A more efficient formula arises if we add the area of a new trapezoid to the previous integral:

$$v(t_k) = v(t_{k-1}) + \int_{t_{k-1}}^{t_k} a(\tau)d\tau \approx v(t_{k-1}) + \Delta t\frac{1}{2}(a_{k-1} + a_k), \qquad (6.3)$$

for $k = 1, 2, ldots, n-1$, while $v_0 = 0$. Use this formula to fill an array `v` with velocity values. Now only $\Delta t$ is given on the command line, and the $a_0, \ldots, a_{n-1}$ values must be read from file as in Exercise 6.5. Name of program file: `acc2vel.py`. ◇

**Exercise 6.7.** *Find velocity from GPS coordinates.*

Imagine that a GPS device measures your position at every $s$ seconds. The positions are stored as $(x, y)$ coordinates in a file `src/files/pos.dat` with the an $x$ and $y$ number on each line, except for the first line which contains the value of $s$.

First, load $s$ into a `float` variable and the $x$ and $y$ numbers into two arrays and draw a straight line between the points (i.e., plot the $y$ coordinates versus the $x$ coordinates).

The next task is to compute and plot the velocity of the movements. If $x(t)$ and $y(t)$ are the coordinates of the positions as a function of time, we have that the velocity in $x$ direction is $v_x(t) = dx/dt$, and the velocity in $y$ direction is $v_y = dy/dt$. Since $x$ and $y$ are only known for some discrete times, $t_k = ks$, $k = 0, \ldots, n-1$, we must use numerical differentation. A simple (forward) formula is

$$v_x(t_k) \approx \frac{x(t_{k+1}) - x(t_k)}{s}, \quad v_y(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{s}, \quad k = 0, \ldots, n-2\,.$$

Compute arrays `vx` and `vy` with velocities based on the formulas above for $v_x(t_k)$ and $v_y(t_k)$, $k = 0, \ldots, n-2$. Plot `vx` versus time and `vy` versus time. Name of program file: `position2velocity.py`. ◇

**Exercise 6.8.** *Make a dictionary from a table.*

The file `src/files/constants.txt` contains a table of the values and the dimensions of some fundamental constants from physics. We want to load this table into a dictionary `constants`, where the keys are

the names of the constants. For example, `constants['gravitational constant']` holds the value of the gravitational constant ($6.67259 \cdot 10^{-11}$) in Newton's law of gravitation. You may either initialize the dictionary by a program that reads and interprets the text in the file, or you may manually cut and paste text in the file into a program where you define the dictionary. Name of program file: `fundamental_constants.py`. ◇

**Exercise 6.9.** *Explore syntax differences: lists vs. dictionaries.*

Consider this code:

```
t1 = {}
t1[0] = -5
t1[1] = 10.5
```

Explain why the lines above work fine while the ones below do not:

```
t2 = []
t2[0] = -5
t2[1] = 10.5
```

What must be done in the last code snippet to make it work properly? Name of program file: `list_vs_dict.py`. ◇

**Exercise 6.10.** *Improve the program from Ch. 6.2.4.*

Consider the program `density.py` from Chapter 6.2.4. One problem we face when implementing this program is that the name of the substance can contain one or two words, and maybe more words in a more comprehensive table. The purpose of this exercise is to use string operations to shorten the code and make it more general. Implement the following two methods in separate functions in the same program, and control that they give the same result.

1. Let `substance` consist of all the words but the last, using the `join` method in string objects to combine the words.
2. Observe that all the densities start in the same column file and use substrings to divide `line` into two parts. (Hint: Remember to strip the first part such that, e.g., the density of ice is obtained as `densities['ice']` and not `densities['ice          ']`.)

Name of program file: `density_improved.py`. ◇

**Exercise 6.11.** *Interpret output from a program.*

The program `src/basic/lnsum.py` produces, among other things, this output:

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

Redirect the output to a file. Write a Python program that reads the file and extracts the numbers corresponding to `epsilon`, `exact error`,

and `n`. Store the numbers in three arrays and plot `epsilon` and the `exact error` versus `n`. Use a logarithmic scale on the $y$ axis, which is enabled by the `log='y'` keyword argument to the `plot` function. Name of program file: `read_error.py`. ◇

**Exercise 6.12.** *Make a dictionary.*

Based on the stars data in Exercise 2.44, make a dictionary where the keys contain the names of the stars and the values correspond to the luminosity. Name of program file: `stars_data_dict1.py`. ◇

**Exercise 6.13.** *Make a nested dictionary.*

Store the data about stars from Exercise 2.44 in a nested dictionary such that we can look up the distance, the apparent brightness, and the luminosity of a star with name `N` by `stars[N]['distance']`, `stars[N]['apparent brightness']`, and `stars[N]['luminosity']`. Name of program file: `stars_data_dict2.py`. ◇

**Exercise 6.14.** *Make a nested dictionary from a file.*

The file `src/files/human_evolution.txt` holds information about various human species and their hight, weight, and brain volume. Make a program that reads this file and stores the tabular data in a nested dictionary `humans`. The keys in `humans` correspond to the specie name (e.g., "homo erectus"), and the values are dictionaries with keys for "height", "weight", "brain volume", and "when" (the latter for when the specie lived). For example, `humans['homo neanderthalensis']['mass']` should equal `'55-70'`. Let the program write out the `humans` dictionary in a nice tabular form similar to that in the file. Name of program file: `humans.py`. ◇

**Exercise 6.15.** *Compute the area of a triangle.*

The purpose of this exercise is to write an `area` function as in Exercise 2.17, but now we assume that the vertices of the triangle is stored in a dictionary and not a list. The keys in the dictionary correspond to the vertex number (1, 2, or 3) while the values are 2-tuples with the $x$ and $y$ coordinates of the vertex. For example, in a triangle with vertices $(0, 0)$, $(1, 0)$, and $(0, 2)$ the `vertices` argument becomes

```
{1: (0,0), 2: (1,0), 3: (0,2)}
```

Name of program file: `area_triangle_dict.py`. ◇

**Exercise 6.16.** *Compare data structures for polynomials.*

Write a code snippet that uses both a list and a dictionary to represent the polynomial $-\frac{1}{2} + 2x^{100}$. Print the list and the dictionary, and use them to evaluate the polynomial for $x = 1.05$ (you can apply the `poly1` and `poly2` functions from Chapter 6.2.3). Name of program file: `poly_repr.py`. ◇

**Exercise 6.17.** *Compute the derivative of a polynomial.*

A polynomial can be represented by a dictionary as explained in Chapter 6.2.3. Write a function `diff` for differentiating such a polynomial. The `diff` function takes the polynomial as a dictionary argument and returns the dictionary representation of the derivative. Recall the formula for differentiation of polynomials:

$$\frac{d}{dx}\sum_{j=0}^{n} c_j x^j = \sum_{j=1}^{n} j c_j x^{j-1}\,. \tag{6.4}$$

This means that the coefficient of the $x^{j-1}$ term in the derivative equals $j$ times the coefficient of $x^j$ term of the original polynomial. With `p` as the polynomial dictionary and `dp` as the dictionary representing the derivative, we then have `dp[j-1] = k*p[j]` for `j` running over all keys in `p`, except when `j` equals 0.

Here is an example of the use of the function `diff`:

```
>>> p = {0: -3, 3: 2, 5: -1}    # -3 + 2*x**3 - x**5
>>> diff(p)                     # should be 6*x**2 - 5*x**4
{2: 6, 4: -5}
```

Name of program file: `poly_diff.py`. ◇

**Exercise 6.18.** *Generalize the program from Ch. 6.2.6.*

The program from Chapter 6.2.6 is specialized for three particular companies. Suppose you download $n$ files from *finance.yahoo.com*, all with monthly stock price data for the *same* period of time. Also suppose you name these files `company.csv`, where `company` reflects the name of the company. Modify the program from Chapter 6.2.6 such that it reads a set of filenames from the command line and creates a plot that compares the evolution of the corresponding stock prices. Normalize all prices such that they initially start at a unit value. Name of program file: `stockprices3.py`. ◇

**Exercise 6.19.** *Write function data to file.*

We want to dump $x$ and $f(x)$ values to a file, where the $x$ values appear in the first column and the $f(x)$ values appear in the second. Choose $n$ equally spaced $x$ values in the interval $[a, b]$. Provide $f$, $a$, $b$, $n$, and the filename as input data on the command line. Use the `StringFunction` tool (see Chapters 3.1.4 and 4.4.3) to turn the textual expression for $f$ into a Python function. (Note that the program from Exercise 6.1 can be used to read the file generated in the present exercise into arrays again for visualization of the curve $y = f(x)$.) Name of program files `write_cml_function.py`. ◇

**Exercise 6.20.** *Specify functions on the command line.*

Explain what the following two code snippets do and give an example of how they can be used. Snippet 1:

```
import sys
from scitools.StringFunction import StringFunction
parameters = {}
for prm in sys.argv[4:]:
    key, value = prm.split('=')
    parameters[key] = eval(value)
f = StringFunction(sys.argv[1], independent_variables=sys.argv[2],
                   **parameters)
var = float(sys.argv[3])
print f(var)
```

Snippet 2:

```
import sys
from scitools.StringFunction import StringFunction
f = eval('StringFunction(sys.argv[1], ' + \
         'independent_variables=sys.argv[2], %s)' % \
         (', '.join(sys.argv[4:])))
var = float(sys.argv[3])
print f(var)
```

Hint: Read about the `StringFunction` tool in Chapter 3.1.4 and about a variable number of keyword arguments in Appendix E.5. Name of program file: `cml_functions.py`. ◇

**Exercise 6.21.** *Interpret function specifications.*

To specify arbitrary functions $f(x_1, x_2, \ldots; p_1, p_2, \ldots)$ with independent variables $x_1, x_2, \ldots$ and a set of parameters $p_1, p_2, \ldots$, we allow the following syntax on the command line or in a file:

```
<expression> is function of <list1> with parameter <list2>
```

where `<expression>` denotes the function formula, `<list1>` is a comma-separated list of the independent variables, and `<list2>` is a comma-separated list of name=value parameters. The part `with parameters <list2>` is omitted if there are no parameters. The names of the independent variables and the parameters can be chosen freely as long as the names can be used as Python variables. Here are some examples of this syntax can be used to specify:

```
sin(x) is a function of x
sin(a*y) is a function of y with parameter a=2
sin(a*x-phi) is a function of x with parameter a=3, phi=-pi
exp(-a*x)*cos(w*t) is a function of t with parameter a=1,w=pi,x=2
```

Create a Python function that takes such function specifications as input and returns an appropriate `StringFunction` object. This object must be created from the function expression and the list of independent variables and parameters. For example, the last function specification above leads to the following `StringFunction` creation:

```
f = StringFunction('exp(-a*x)*sin(k*x-w*t)',
                   independent_variables=['t'],
                   a=1, w=pi, x=2)
```

Hint: Use string operations to extract the various parts of the string. For example, the expression can be split out by calling `split('is a`

function'). Typically, you need to extract <expression>, <list1>, and <list2>, and create a string like

```
StringFunction(<expression>, independent_variables=[<list1>],
               <list2>)
```

and sending it to eval to create the object. Name of program file: text2func.py. ◇

**Exercise 6.22.** *Compare average temperatures in two cities.*

Chapter 6.4 exemplifies how we can extract temperature data from the web. Similar data for the city of Stockholm is available at

```
ftp://ftp.engr.udayton.edu/jkissock/gsod/SNSTKHLM.txt
```

If we inspect the *.txt files containing temperature data from Oslo and Stockholm, we observe that even though most of the temperatures seem reasonable, the value -99 keeps appearing as a temperature. This value indicates a missing observation, and the value must not enter the computations of the average temperatures.

Make a Python program that computes the average temperature in Celsius degrees since 1995 in Oslo and Stockholm. Hint: You do not need to store the data in a dictionary as in Chapter 6.4 – adding up the numbers in the 4th column is sufficient. Name of program file: compare_mean_temp.py. ◇

**Exercise 6.23.** *Compare average temperatures in many cities.*

You should do Exercise 6.22 first. The URL

```
http://www.engr.udayton.edu/weather/citylistWorld.htm
```

contains links to temperature data for many cities around the world. The task of this exercise is to make a list of the cities and their average temperatures since 1995 and until the present date. Your program will download the files containing the temperature data from all the cities listed on the web page. This may be a rather long process, so make sure you have quite some time available. Fortunately, you only need to download all the files once. Use the test

```
if os.path.isfile(filename):
```

to check if a particular file with name filename is already present in the current folder, so you can avoid a new download.

First, you need to interpret the HTML text in the citylistWorld.htm file whose complete URL is given above. Download the file, inspect it, and realize that there are two types of lines of interest, one with the city name, typically on the form,

```
mso-list:l6 level1 lfo3;tab-stops:list .5in'><b>Algiers ( </b><b...
```

and one with the URL of the temperature data,

```
href="ftp://ftp.engr.udayton.edu/jkissock/gsod/ALALGIER.txt">ALA...
```

The first one can be detected by a test `line.find(" .5in'>")`, if `line` is a string containing a line from the file. Extracting the city name can be performed by, for example, a `line.split(">")` and picking the right component, and stripping off leading and trailing characters. The URL for the temperature data appears some lines below the city name. The line can be found by using `line.find("href")`. One can extract the URL by splitting with respect to `'>'` and stripping off the initial `href="` text.

Store the city names and the associated URLs in a dictionary. Thereafter, go through all the cities and compute their average temperature values. You can change the values of the dictionary to store both the temperature value and the URL as a 2-tuple.

Finally, write out the cities and their average temperatures in sorted sequence, starting with the hottest city and ending with the coolest. To sort the dictionary, first transform it to a list of 3-tuples (cityname, URL, temperature), and then write a tailored sort function for this type of list elements (see Exercise 2.44 for details about a similar tailored sort function). Make sure that the values `-99` for missing data do not enter any computations.

Organize the program as a module (Chapter 3.5), i.e., a set of functions for carrying out the main steps and a test block where the functions are called. Some of the functionality in this module can be reused in Exercises 6.24 and 6.25. Name of program file: `sort_mean_temp_cities.py`. ◇

**Exercise 6.24.** *Plot the temperature in a city, 1995-today.*

The purpose of this exercise is to read the name of a city from the command line, and thereafter present a plot of the temperature in that city from 1995 to today. You must first carry out Exercise 6.23 so that you have a module with a function that returns a dictionary with city names as keys and the corresponding URL for the temperature data files as values. The URL must be opened, and the temperature data must be read into an array. We plot this array against its indices, not against year, month, and day (that will be too complicated). Note that the temperature data may contain values -99, indicating missing recordings, and these values will lead to wrong, sudden jumps in the plots. If you insert the value `NaN` (a NumPy type for representing "Not a Number") instead of the numerical values -99 in arrays, some plotting programs will plot the array correctly, i.e., as several curve segments where the missing data are left out. This is true if you use Easyviz with Gnuplot as plotting program.

Construct the program as a module, where there are two functions that can be imported in other programs:

```
def get_city_URLs():
    """Return dictionary d[cityname] = URL."""

def get_temperatures(URL):
    """Return array with temperature values read from URL."""
```

Name of program file: `plot_temp.py`. ◇

**Exercise 6.25.** *Plot temperatures in several cities.*
This exercise is a continuation of Exercise 6.24. Make a program that starts with printing out the names of all cities for which we have temperature data from 1995 to today. Then ask the user for the names of some cities (separated by blanks). Thereafter load the temperature data for these cities (use the `get_city_URLs` and `get_temperatures` functions from Exercise 6.24) and visualize them in the same plot. Name of program file: `plot_multiple_temps.py`. ◇

**Exercise 6.26.** *Try Word or OpenOffice to write a program.*
The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

Type the following one-line program in either Microsoft Word or OpenOffice:

```
print "Hello, World!"
```

Both Word and OpenOffice are so "smart" that they automatically edit "print" to "Print" since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

Save the program as a `.doc` (Word) or `.odt` (OpenOffice) file. Now try to run this file as a Python program. You will get a message

```
SyntaxError: Non-ASCII character
```

Explain why you get this error.

Then save the program as a `.txt` file. Run this file as a Python program. It may work well if you wrote the program text in Microsoft Word, but with OpenOffice there may still be strange characters in the file. Use a text editor to view the exact contents of the file. Name of program file: `office.py`. ◇

**Exercise 6.27.** *Evaluate objects in a boolean context.*
Writing `if a:` or `while a:` in a program, where `a` is some object, requires evaluation of `a` in a boolean context. To see the value of an object `a` in a boolean context, one can call `bool(a)`. Try the following program to learn what values of what objects that are `True` or `False` in a boolean context:

```
objects = [
  '""',               # empty string
  '"string"',         # non-empty string
  '[]',               # empty list
  '[0]',              # list with one element
  '()',               # empty tuple
  '(0,)',             # tuple with one element
  '{}',               # empty dict
  '{0:0}',            # dict with one element
  '0',                # int zero
  '0.0',              # float zero
  '0j',               # complex zero
  '10',               # int 10
  '10.',              # float 10
  '10j'               # imaginary 10
  'zeros(0)',         # empty array
  'zeros(1)',         # array with one element (zero)
  'zeros(1)+10',      # array with one element (10)
  'zeros(2)',         # array with two elements (watch out!)
  ]
for element in objects:
    object = eval(element)
    print 'object = %s; if object: is %s' % \
        (element, bool(object))
```

Write down a rule for the family of Python objects that evaluate to `False` in a boolean context. ◇

**Exercise 6.28.** *Generate an HTML report.*

Extend the program made in Exercise 5.22 with a report containing all the plots. The report can be written in HTML and displayed by a web browser. The plots must then be generated in PNG format. The source of the HTML file will typically look as follows:

```
<html>
<body>
<p><img src="tmp_y0_0.01_q_0.1_N_50.png">
<p><img src="tmp_y0_0.01_q_1_N_50.png">
<p><img src="tmp_y0_0.01_q_1.5_N_50.png">
<p><img src="tmp_y0_0.01_q_1.8_N_50.png">
...
<p><img src="tmp_y0_0.01_q_3_N_1000.png">
</html>
</body>
```

Let the program write out the HTML text. You can let the function making the plots return the name of the plotfile, such that this string can be inserted in the HTML file. Name of program file: `growth_logistic4.py`. ◇

**Exercise 6.29.** *Fit a polynomial to experimental data.*

Suppose we have measured the oscillation period $T$ of a simple pendulum with a mass $m$ at the end of a massless rod of length $L$. We have varied $L$ and recorded the corresponding $T$ value. The measurements are found in a file `src/files/pendulum.dat`, containing two columns. The first column contains $L$ values and the second column has the corresponding $T$ values.

Load the $L$ and $T$ values into two arrays. Plot $L$ versus $T$ using circles for the data points. We shall assume that $L$ as a function of

$T$ is a polynomial. Use the NumPy utilities `polyfit` and `poly1d`, as explained in Exercise 6.4, and experiment with fitting polynomials of degree 1, 2, and 3. Visualize the polynomial curves together with the experimental data. Which polynomial fits the measured data best? Name of program file: `fit_pendulum_data.py`. ◇

**Exercise 6.30.** *Interpret an HTML file with rainfall data.*

The file `src/files/rainfall.url` contains the URL to several web pages with average rainfall data from major cities in the world. The goal of this exercise is to download all these web pages, and for each web page load the rainfall data into an array where the first 12 elements corresponds to the rainfall in each month of the year, and the 13th element contains the total rainfall in a year.

Make a function `getdata(url)` which downloads a web page with address `url` and returns the name of the weather station (usually a city) and an array with 13 elements containing the average rainfall data found in the web page. Make another function `plotdata(data, location)` which plots the array returned from `getdata` with the `location` of the weather station as plot title. Let `plotdata` make a hardcopy with `location` as a part of the filename (some `location` names in the web pages contain a slash or other characters that are not appropriate in filenames – remove these). Call `getdata` and `plotdata` for each for each of the URLs in the `rainfall.url` file.

Hint: The rainfall data in the web pages appear in an HTML table. The relevant line in the file starts like

```
<tr><td> mm <td align=right>193.0 <td align=right>143.6
```

Assuming that the line is available as the string `line` in the program, a test `if line.startswith('<tr><td> mm')` makes you pick out the right line. You can then strip off the `mm` column by `line[12:]`. Then you can replace `<td align=right>` by an empty string. The line ends with `<br>`, which must be removed. The result is a line with numbers, the monthly and annual rainfall, separated by blanks. A split operation and conversion to `float` creates a list of the data.

Regarding, the location of the weather station, this is found in a line which starts out as

```
<p>Weather station <strong>OSLO/BLINDERN</strong>
```

You can, for example, use `line.find` method to locate the tags `<strong>` and `/strong` and thereby extract the location of the weather station.

Name of program file: `download_rainfall_data.py`. ◇

**Exercise 6.31.** *Generate an HTML report with figures.*

The goal of this exercise is to let a program write a report in HTML format. The report starts with the Python code for the $f(x,t)$ function from Exercise 4.17 on page 229. Program code can be placed inside `<pre>` and `</pre>` tags. The report should continue with three plots of

the function in Exercise 4.11 for three different $t$ values (find suitable $t$ values that illustrate the displacement of the wave packet). At the end, there is an animated GIF file with the movie from Exercise 4.17. Inserte appropriate headlines (`<h1>` tags) in the report. Name of program file: `wavepacket_report.py`. ◇

# Introduction to Classes 7

A class packs a set of data (variables) together with a set of functions operating on the data. The goal is to achieve more modular code by grouping data and functions into manageable (often small) units. Most of the mathematical computations in this book can easily be coded without using classes, but in many problems, classes enable either more elegant solutions or code that is easier to extend at a later stage. In the non-mathematical world, where there are no mathematical concepts and associated algorithms to help structure the problem solving, software development can be very challenging. Classes may then improve the understanding of the problem and contribute to simplify the modeling of data and actions in programs. As a consequence, almost all large software systems being developed in the world today are heavily based on classes.

Programming with classes is offered by most modern programming languages, also Python. In fact, Python employs classes to a very large extent, but one can – as we have seen in previous chapters – use the language for lots of purposes without knowing what a class is. However, one will frequently encounter the class concept when searching books or the World Wide Web for Python programming information. And more important, classes often provide better solutions to programming problems. This chapter therefore gives an introduction to the class concept with emphasis on applications to numerical computing. More advanced use of classes, including inheritance and object orientation, is the subject of Chapter 9.

The folder `src/class` contains all the program examples from the present chapter.

## 7.1 Simple Function Classes

Classes can be used for many things in scientific computations, but one of the most frequent programming tasks is to represent mathematical functions which have a set of parameters in addition to one or more independent variables. Chapter 7.1.1 explains why such mathematical functions pose difficulties for programmers, and Chapter 7.1.2 shows how the class idea meets these difficulties. Chapters 7.1.3 presents another example where a class represents a mathematical function. More advanced material about classes, which for some readers may clarify the ideas, but which can also be skipped in a first reading, appears in Chapters 7.1.4 and Chapter 7.1.5.

### 7.1.1 Problem: Functions with Parameters

To motivate for the class concept, we will look at functions with parameters. The $y(t) = v_0 t - \frac{1}{2}gt^2$ function on page 1 is such a function. Conceptually, in physics, $y$ is a function of $t$, but $y$ also depends on two other parameters, $v_0$ and $g$, although it is not natural to view $y$ as a function of these parameters. We may write $y(t; v_0, g)$ to indicate that $t$ is the independent variable, while $v_0$ and $g$ are parameters. Strictly speaking, $g$ is a fixed parameter[1], so only $v_0$ and $t$ can be arbitrarily chosen in the formula. It would then be better to write $y(t; v_0)$.

In the general case, we may have a function of $x$ that has $n$ parameters $p_1, \ldots, p_n$: $f(x; p_1, \ldots, p_n)$. One example could be

$$g(x; A, a) = Ae^{-ax}\,.$$

How should we implement such functions? One obvious way is to have the independent variable and the parameters as arguments:

```
def y(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(x, a, A):
    return A*exp(-a*x)
```

*Problem.* There is one major problem with this solution. Many software tools we can use for mathematical operations on functions assume that a function of one variable has only one argument in the computer representation of the function. For example, we may have a tool for differentiating a function $f(x)$ at a point $x$, using the approximation

[1] As long as we are on the surface of the earth, $g$ can be considered fixed, but in general $g$ depends on the distance to the center of the earth.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{7.1}$$

coded as

```
def diff(f, x, h=1E-10):
    return (f(x+h) - f(x))/h
```

The `diff` function works with any function `f` that takes one argument:

```
def h(t):
    return t**4 + 4*t

dh = diff(h, 0.1)

from math import sin, pi
x = 2*pi
dsin = diff(sin, x, h=1E-9)
```

Unfortunately, `diff` will not work with our `y(t, v0)` function. Calling `diff(y, t)` leads to an error inside the `diff` function, because it tries to call our `y` function with only one argument while the `y` function requires two.

Writing an alternative `diff` function for `f` functions having two arguments is a bad remedy as it restricts the set of admissible `f` functions to the very special case of a function with one independent variable and one parameter. A fundamental principle in computer programming is to strive for software that is as general and widely applicable as possible. In the present case, it means that the `diff` function should be applicable to all functions `f` of one variable, and letting `f` take one argument is then the natural decision to make.

The mismatch of function arguments, as outlined above, is a major problem because a lot of software libraries are available for operations on mathematical functions of one variable: integration, differentiation, solving $f(x) = 0$, finding extrema, etc. (see for instance Chapters 3.6.2 and 5.1.9,, and Appendices A and B). All these libraries will try to call the mathematical function we provide with only one argument.

*A Bad Solution: Global Variables.* The requirement is thus to define Python implementations of mathematical functions of one variable with one argument, the independent variable. The two examples above must then be implemented as

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2

def g(t):
    return A*exp(-a*x)
```

These functions work only if `v0`, `A`, and `a` are global variables, initialized before one attempts to call the functions. Here are two sample calls where `diff` differentiates `y` and `g`:

```
v0 = 3
dy = diff(y, 1)

A = 1; a = 0.1
dg = diff(g, 1.5)
```

The use of global variables is in general considered bad programming. Why global variables are problematic in the present case can be illustrated when there is need to work with several versions of a function. Suppose we want to work with two versions of $y(t; v_0)$, one with $v_0 = 1$ and one with $v_0 = 5$. Every time we call `y` we must remember which version of the function we work with, and set `v0` accordingly prior to the call:

```
v0 = 1; r1 = y(t)
v0 = 5; r2 = y(t)
```

Another problem is that variables with simple names like `v0`, `a`, and `A` may easily be used as global variables in other parts of the program. These parts may change our `v0` in a context different from the `y` function, but the change affects the correctness of the `y` function. In such a case, we say that changing `v0` has *side effects*, i.e., the change affects other parts of the program in an unintentional way. This is one reason why a golden rule of programming tells us to limit the use of global variables as much as possible.

Another solution to the problem of needing two $v_0$ parameters could be to introduce two `y` functions, each with a distinct $v_0$ parameter:

```
def y1(t):
    g = 9.81
    return v0_1*t - 0.5*g*t**2

def y2(t):
    g = 9.81
    return v0_2*t - 0.5*g*t**2
```

Now we need to initialize `v0_1` and `v0_2` once, and then we can work with `y1` and `y2`. However, if we need 100 $v_0$ parameters, we need 100 functions. This is tedious to code, error prone, difficult to administer, and simply a really bad solution to a programming problem.

So, is there a good remedy? The answer is yes: The class concept solves all the problems described above!

### 7.1.2 Representing a Function as a Class

A class contains a set of variables (data) and a set of functions, held together as one unit. The variables are visible in all the functions in the class. That is, we can view the variables as "global" in these functions. These characteristics also apply to modules, and modules can be used to obtain many of the same advantages as classes offer (see comments

in Chapter 7.1.5). However, classes are technically very different from modules. You can also make many copies of a class, while there can be only one copy of a module. When you master both modules and classes, you will clearly see the similarities and differences. Now we continue with a specific example of a class.

Consider the function $y(t; v_0) = v_0 t - \frac{1}{2}gt^2$. We may say that $v_0$ and $g$, represented by the variables `v0` and `g`, constitute the data. A Python function, say `value(t)`, is needed to compute the value of $y(t; v_0)$ and this function must have access to the data `v0` and `g`, while `t` is an argument.

A programmer experienced with classes will then suggest to collect the data `v0` and `g`, and the function `value(t)`, together as a class. In addition, a class usually has another function, called *constructor* for initializing the data. The constructor is always named `__init__`. Every class must have a name, often starting with a capital, so we choose `Y` as the name since the class represents a mathematical function with name $y$. Figure 7.1 sketches the contents of class `Y` as a so-called UML diagram, here created with Lumpy (from Appendix E.3) with aid of the little program `class_Y_v1_UML.py`. The UML diagram has two “boxes”, one where the functions are listed, and one where the variables are listed. Our next step is to implement this class in Python.

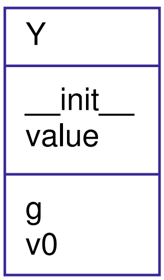



**Fig. 7.1** UML diagram with function and data in the simple class `Y` for representing a mathematical function $y(t; v_0)$.

*Implementation.* The complete code for our class `Y` looks as follows in Python:

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

A puzzlement for newcomers to Python classes is the `self` parameter, which may take some efforts and time to fully understand.

*Usage and Dissection.* Before we dig into what each in the class implementation means, we start by showing how the class can be used to compute values of the mathematical function $y(t; v_0)$.

A class creates a new data type, so now we have a data type `Y` of which we can make objects of[2]. An object of a user-defined class (like `Y`) is usually called an *instance*. We need such an instance in order to use the data in the class and call the `value` function. An object of a user-defined class (like `Y`) is usually called an *instance*. The following statement constructs an instance bound to the variable name `y`:

```
y = Y(3)
```

Seemingly, we call the class `Y` as if it were a function. Actually, `Y(3)` is automatically translated by Python to a call to the constructor `__init__` in class `Y`. The arguments in the call, here only the number `3`, are always passed on as arguments to `__init__` *after* the `self` argument. That is, `v0` gets the value `3` and `self` is just dropped in the call. This may be confusing, but it is a rule that the `self` argument is never used in calls to functions in classes.

With the instance `y`, we can compute the value $y(t = 0.1; v_0 = 3)$ by the statement

```
v = y.value(0.1)
```

Here also, the `self` argument is dropped in the call to `value`. To access functions and variables in a class, we must prefix the function and variable names by the name of the instance and a dot: the `value` function is reached as `y.value`, and the variables are reached as `y.v0` and `y.g`. We can, for example, print the value of `v0` in the instance `y` by writing

```
print y.v0
```

The output will in this case be `3`.

We have already introduced the term "instance" for the object of a class. Functions in classes are commonly called *methods*, and variables (data) in classes are called *attributes*. From now on we will use this terminology. In our sample class `Y` we have two methods, `__init__` and `value`, and two attributes, `v0` and `g`. The names of methods and attributes can be chosen freely, just as names of ordinary Python functions and variables. However, the constructor must have the name `__init__`, otherwise it is not automatically called when we create new instances.

You can do whatever you want in whatever method, but it is a convention to use the constructor for initializing the variables in the class such that the class is "ready for use".

[2] All familiar Python objects, like lists, tuples, strings, floating-point numbers, integers, etc., are in fact built-in Python classes, with names `list`, `tuple`, `str`, `float`, `int`, etc.

*The* `self` *Variable.* Now we will provide some more explanation of the `self` parameter and how the class methods work. Inside the constructor `__init__`, the argument `self` is a variable holding the new instance to be constructed. When we write

```
    self.v0 = v0
    self.g = 9.81
```

we define two new attributes in this instance. The `self` parameter is invisibly returned to the calling code. We can imagine that Python translates `y = Y(3)` to

```
Y.__init__(y, 3)
```

so when we do a `self.v0 = v0` in the constructor, we actually initialize `y.v0`. The prefix with `Y.` is necessary to reach a class method (just like prefixing a function in a module with the module name, e.g., `math.exp`). If we prefix with `Y.`, we need to explicitly feed in an instance for the `self` argument, like `y` in the code line above, but if we prefix with `y.` (the instance name) the `self` argument is dropped. It is the latter "instance name prefix" which we shall use when computing with classes.

Let us look at a call to the `value` method to see a similar use of the `self` argument. When we write

```
value = y.value(0.1)
```

Python translates this to a call

```
value = Y.value(y, 0.1)
```

such that the `self` argument in the `value` method becomes the `y` instance. In the expression inside the `value` method,

```
self.v0*t - 0.5*self.g*t**2
```

`self` is `y` so this is the same as

```
y.v0*t - 0.5*y.g*t**2
```

The rules regarding "`self`" are listed below:

- Any class method must have `self` as first argument[3].
- `self` represents an (arbitrary) instance of the class.
- To access another class method or a class attribute, inside class methods, we must prefix with `self`, as in `self.name`, where `name` is the name of the attribute or the other method.
- `self` is dropped as argument in calls to class methods.

[3] The name can be any valid variable name, but the name `self` is a widely established convention in Python.

It takes some time to understand the `self` variable, but more examples and hands-on experience with class programming will help, so just be patient and continue reading.

*Extension of the Class.* We can have as many attributes and methods as we like in a class, so let us add a new method to class `Y`. This method is called `formula` and prints a string containing the formula of the mathematical function $y$. After this formula, we provide the value of $v_0$. The string can then be constructed as

```
'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

where `self` is an instance of class `Y`. A call of `formula` does not need any arguments:

```
print y.formula()
```

should be enough to create, return, and print the string. However, even if the `formula` method does not need any arguments, it must have a `self` argument, which is left out in the call but needed inside the method to access the attributes. The implementation of the method is therefore

```
    def formula(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

For completeness, the whole class now reads

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def formula(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Example on use may be

```
y = Y(5)
t = 0.2
v = y.value(t)
print 'y(t=%g; v0=%g) = %g' % (t, y.v0, v)
print y.formula()
```

with the output

```
y(t=0.2; v0=5) = 0.8038
v0*t - 0.5*g*t**2; v0=5
```

*Remark.* A common mistake done by newcomers to the class construction is to place the code that applies the class at the same indentation as the class methods. This is illegal. Only method definitions and assignments to so-called static attributes (Chapter 7.7) can appear in

the indented block under the `class` headline. Ordinary attribute assignment must be done inside methods. The main program using the class must appear with the same indent as the `class` headline.

*Using Methods as Ordinary Functions.* We may create several $y$ functions with different values of $v_0$:

```
y1 = Y(1)
y2 = Y(1.5)
y3 = Y(-3)
```

We can treat `y1.value`, `y2.value`, and `y3.value` as ordinary Python functions of `t`, and then pass them on to any Python function that expects a function of one variable. In particular, we can send the functions to the `diff(f, x)` function from page 339:

```
dy1dt = diff(y1.value, 0.1)
dy2dt = diff(y2.value, 0.1)
dy3dt = diff(y3.value, 0.2)
```

Inside the `diff(f, x)` function, the argument `f` now behaves as a function of one variable that automatically carries with it two variables `v0` and `g`. When `f` refers to (e.g.) `y3.value`, Python actually knows that `f(x)` means `y3.value(x)`, and inside the `y3.value` method `self` is `y3`, and we have access to `y3.v0` and `y3.g`.

*Doc Strings.* A function may have a doc string right after the function definition, see Chapter 2.2.7. The aim of the doc string is to explain the purpose of the function and, for instance, what the arguments and return values are. A class can also have a doc string, it is just the first string that appears right after the `class` headline. The convention is to enclose the doc string in triple double quotes `"""`:

```
class Y:
    """The vertical motion of a ball."""

    def __init__(self, v0):
        ...
```

More comprehensive information can include the methods and how the class is used in an interactive session:

```
class Y:
    """
    Mathematical function for the vertical motion of a ball.

    Methods:
       constructor(v0): set initial velocity v0.
       value(t): compute the height as function of t.
       formula(): print out the formula for the height.

    Attributes:
       v0: the initial velocity of the ball (time 0).
       g: acceleration of gravity (fixed).
```

```
    Usage:
    >>> y = Y(3)
    >>> position1 = y.value(0.1)
    >>> position2 = y.value(0.3)
    >>> print y.formula()
    v0*t - 0.5*g*t**2; v0=3
    """
```

### 7.1.3 Another Function Class Example

Let us apply the ideas from the `Y` class to the $v(r)$ function specified in (4.20) on page 230. We may write this function as $v(r; \beta, \mu_0, n, R)$ to indicate that there is one primary independent variable ($r$) and four physical parameters ($\beta$, $\mu_0$, $n$, and $R$). The class typically holds the physical parameters as variables and provides an `value(r)` method for computing the $v$ function:

```
class VelocityProfile:
    def __init__(self, beta, mu0, n, R):
        self.beta, self.mu0, self.n, self.R = beta, mu0, n, R

    def value(self, r):
        beta, mu0, n, R = self.beta, self.mu0, self.n, self.R
        n = float(n)  # ensure float divisions
        v = (beta/(2.0*mu0))**(1/n)*(n/(n+1))*\
            (R**(1+1/n) - r**(1+1/n))
        return v
```

There is seemingly one new thing here in that we initialize several variables on the same line[4]:

```
        self.beta, self.mu0, self.n, self.R = beta, mu0, n, R
```

This is perfectly valid Python code and equivalent to the multi-line code

```
        self.beta = beta
        self.mu0 = mu0
        self.n = n
        self.R = R
```

In the `value` method it is convenient to avoid the `self.` prefix in the mathematical formulas and instead introduce the local short names `beta`, `mu0`, `n`, and `R`. This is in general a good idea, because it makes it easier to read the implementation of the formula and check its correctness.

Here is one possible application of class `VelocityProfile`:

[4] The comma-separated list of variables on the right-hand side forms a tuple so this assignment is just the usual construction where a set of variables on the left-hand side is set equal to a list or tuple on the right-hand side, element by element. See page 58.

```
v1 = VelocityProfile(R=1, beta=0.06, mu0=0.02, n=0.1)
# plot v1 versus r:
from scitools.std import *
r = linspace(0, 1, 50)
v = v1.value(r)
plot(r, v, label=('r', 'v'), title='Velocity profile')
```

*Remark.* Another solution to the problem of sending functions with parameters to a general library function such as `diff` is provided in Appendix E.5. The remedy there is to transfer the parameters as arguments "through" the `diff` function. This can be done in a general way as explained in that appendix.

### 7.1.4 Alternative Function Class Implementations

To illustrate class programming further, we will now realize class `Y` from Chapter 7.1.2 in a different way. You may consider this section as advanced and skip it, but for some readers the material might improve the understanding of class `Y` and give some insight into class programming in general.

It is a good habit always to have a constructor in a class and to initialize class attributes here, but this is not a requirement. Let us drop the constructor and make `v0` an optional argument to the `value` method. If the user does not provide `v0` in the call to `value`, we use a `v0` value that must have been provided in an earlier call and stored as an attribute `self.v0`. We can recognize if the user provides `v0` as argument or not by using `None` as default value for the keyword argument and then test if `v0 is None`.

Our alternative implementation of class `Y`, named `Y2`, now reads

```
class Y2:
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        return self.v0*t - 0.5*g*t**2
```

This time the class has only one method and one attribute as we skipped the constructor and let `g` be a local variable in the `value` method.

But if there is no constructor, how is an instance created? Python fortunately creates an empty constructor. This allows us to write

```
y = Y2()
```

to make an instance `y`. Since nothing happens in the automatically generated empty constructor, `y` has no attributes at this stage. Writing

```
print y.v0
```

therefore leads to the exception

```
AttributeError: Y2 instance has no attribute 'v0'
```

By calling

```
v = y.value(0.1, 5)
```

we create an attribute `self.v0` insde the `value` method. In general, we can create any attribute `name` in any method by just assigning a value to `self.name`. Now trying a

```
print y.v0
```

will print `5`. In a new call,

```
v = y.value(0.2)
```

the previous `v0` value (5) is used inside `value` as `self.v0` unless a `v0` argument is specified in the call.

The previous implementation is not foolproof if we fail to initialize `v0`. For example, the code

```
y = Y2()
v = y.value(0.1)
```

will terminate in the `value` method with the exception

```
AttributeError: Y2 instance has no attribute 'v0'
```

As usual, it is better to notify the user with a more informative message. To check if we have an attribute `v0`, we can use the Python function `hasattr`. Calling `hasattr(self, 'v0')` returns `True` only if the instance `self` has an attribute with name `'v0'`. An improved `value` method now reads

```
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        if not hasattr(self, 'v0'):
            print 'You cannot call value(t) without first '\
                  'calling value(t,v0) to set v0'
            return None
        g = 9.81
        return self.v0*t - 0.5*g*t**2
```

Alternatively, we can try to access `self.v0` in a `try-except` block, and perhaps raise an exception `TypeError` (which is what Python raises if there are not enough arguments to a function or method):

```
    def value(self, t, v0=None):
        if v0 is not None:
            self.v0 = v0
        g = 9.81
        try:
            value = self.v0*t - 0.5*g*t**2
        except AttributeError:
            msg = 'You cannot call value(t) without first '
                  'calling value(t,v0) to set v0'
            raise TypeError(msg)
        return value
```

Note that Python detects an `AttributeError`, but from a user's point of view, not enough parameters were supplied in the call so a `TypeError` is more appropriate to communicate back to the calling code.

We think class `Y` is a better implementation than class `Y2`, because the former is simpler. As already mentioned, it is a good habit to include a constructor and set data here rather than "recording data on the fly" as we try to in class `Y2`. The whole purpose of class `Y2` is just to show that Python provides great flexibility with respect to defining attributes, and that there are no requirements to what a class *must* contain.

### 7.1.5 Making Classes Without the Class Construct

Newcomers to the class conecpt often have a hard time understanding what this concept is about. The present section tries to explain in more detail how we can introduce classes without having the class construct in the computer language. This information may or may not increase your understanding of classes. If not, programming with classes will definitely increase your understanding with time, so there is no reason to worry. In fact, you may safely jump to Chapter 7.3 as there are no important concepts in this section that later sections build upon.

A class contains a collection of variables (data) and a collection of methods (functions). The collection of variables is unique to each instance of the class. That is, if we make ten instances, each of them has its own set of variables. These variables can be thought of as a dictionary with keys equal to the variable names. Each instance then has its own dictionary, and we may roughly view the instance as this dictionary[5].

On the other hand, the methods are shared among the instances. We may think of a method in a class as a standard global function that takes an instance in the form of a dictionary as first argument. The method has then access to the variables in the instance (dictionary) provided in the call. For the `Y` class from Chapter 7.1.2 and an instance

[5] The instance can also contain static class attributes (Chapter 7.7), but these are to be viewed as global variables in the present context.

y, the methods are ordinary functions with the following names and arguments:

```
Y.value(y, t)
Y.formula(y)
```

The class acts as a *namespace*, meaning that all functions must be prefixed by the namespace name, here `Y`. Two different classes, say `C1` and `C2`, may have functions with the same name, say `value`, but when the `value` functions belong to different namespaces, their names `C1.value` and `C2.value` become distinct. Modules are also namespaces for the functions and variables in them (think of `math.sin`, `cmath.sin`, `numpy.sin`).

The only peculiar thing with the class construct in Python is that it allows us to use an alternative syntax for method calls:

```
y.value(t)
y.formula()
```

This syntax coincides with the traditional syntax of calling class methods and providing arguments, as found in other computer languages, such as Java, C#, C++, Simula, and Smalltalk. The dot notation is also used to access variables in an instance such that we inside a method can write `self.v0` instead of `self['v0']` (`self` refers to `y` through the function call).

We could easily implement a simple version of the class concept without having a class construction in the language. All we need is a dictionary type and ordinary functions. The dictionary acts as the instance, and methods are functions that take this dictionary as the first argument such that the function has access to all the variables in the instance. Our `Y` class could now be implemented as

```
def value(self, t):
    return self['v0']*t - 0.5*self['g']*t**2

def formula(self):
    print 'v0*t - 0.5*g*t**2; v0=%g' % self['v0']
```

The two functions are placed in a module called `Y`. The usage goes as follows:

```
import Y
y = {'v0': 4, 'g': 9.81}   # make an "instance"
y1 = Y.value(y, t)
```

We have no constructor since the initialization of the variables is done when declaring the dictionary `y`, but we could well include some initialization function in the `Y` module

```
def init(v0):
    return {'v0': v0, 'g': 9.81}
```

The usage is now slightly different:

```
import Y
y = Y.init(4)       # make an "instance"
y1 = Y.value(y, t)
```

This way of implementing classes with the aid of a dictionary and a set of ordinary functions actually forms the basis for class implementations in many languages. Python and Perl even have a syntax that demonstrates this type of implementation. In fact, every class instance in Python has a dictionary `__dict__` as attribute, which holds all the variables in the instance. Here is a demo that proves the existence of this dictionary in class `Y`:

```
>>> y = Y(1.2)
>>> print y.__dict__
{'v0': 1.2, 'g': 9.8100000000000005}
```

To summarize: A Python class can be thought of as some variables collected in a dictionary, and a set of functions where this dictionary is automatically provided as first argument such that functions always have full access to the class variables.

*First Remark.* We have in this section provided a view of classes *from a technical point of view*. Others may view a class as a way of modeling the world in terms of data and operations on data. However, in sciences that employ the language of mathematics, the modeling of the world is usually done by mathematics, and the mathematical structures provide understanding of the problem and structure of programs. When appropriate, mathematical structures can conveniently be mapped on to classes in programs to make the software simpler and more flexible.

*Second Remark.* The view of classes in this section neglects very important topics such as inheritance and dynamic binding, which we treat in Chapter 9. For more completeness of the present section, we briefly describe how our combination of dictionaries and global functions can deal with inheritance and dynamic binding (but this will not make sense unless you know what inheritance is).

Data inheritance can be obtained by letting a subclass dictionary do an `update` call with the superclass dictionary as argument. In this way all data in the superclass are also available in the subclass dictionary. Dynamic binding of methods is more complicated, but one can think of checking if the method is in the subclass module (using `hasattr`), and if not, one proceeds with checking super class modules until a version of the method is found.

## 7.2 More Examples on Classes

The use of classes to solve problems from mathematical and physical sciences may not be so obvious. On the other hand, in many administrative programs for managing interactions between objects in the real world the objects themselves are natural candidates for being modeled by classes. Below we give some examples on what classes can be used to model.

### 7.2.1 Bank Accounts

The concept of a bank account in a program is a good candidate for a class. The account has some data, typically the name of the account holder, the account number, and the current balance. Three things we can do with an account is withdraw money, put money into the account, and print out the data of the account. These actions are modeled by methods. With a class we can pack the data and actions together into a new data type so that one account corresponds to one variable in a program.

Class `Account` can be implemented as follows:

```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def dump(self):
        s = '%s, %s, balance: %s' % \
            (self.name, self.no, self.balance)
        print s
```

Here is a simple test of how class `Account` can ue used:

```
>>> from classes import Account
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson',  '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print "a1's balance:", a1.balance
a1's balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```

The author of this class does not want users of the class to operate on the attributes directly and thereby change the name, the account

number, or the balance. The intention is that users of the class should only call the constructor, the `deposit`, `withdraw`, and `dump` methods, and (if desired) inspect the `balance` attribute, but never change it. Other languages with class support usually have special keywords that can restrict access to class attributes and methods, but Python does not. Either the author of a Python class has to rely on correct usage, or a special convention can be used: Any name starting with an underscore represents an attribute that should never be touched or a method that should never be called. One refers to names starting with an underscore as *protected* names. These can be freely used inside methods in the class, but not outside.

In class `Account`, it is natural to protect access to the `name`, `no`, and `balance` attributes by prefixing these names by an underscore. For *reading* only of the `balance` attribute, we provide a new method `get_balance`. The user of the class should now only call the methods in the class and not access any attributes.

The new "protected" version of class `Account`, called `AccountP`, reads

```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):
        return self._balance

    def dump(self):
        s = '%s, %s, balance: %s' % \
            (self._name, self._no, self._balance)
        print s
```

We can technically access the attributes, but we then break the convention that names starting with an underscore should never be touched outside the class. Here is class `AccountP` in action:

```
>>> a1 = AccountP('John Olsson', '19371554951', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a1.withdraw(3500)
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> print a1._balance       # it works, but a convention is broken
13500
print a1.get_balance()      # correct way of viewing the balance
13500
>>> a1._no = '19371554955'  # this is a "serious crime"
```

Python has a special construct, called *properties*, that can be used to protect attributes from being changed. This is very useful, but the

author considers properties a bit too complicated for this introductory book.

### 7.2.2 Phone Book

You are probably familiar with the phone book on your mobile phone. The phone book contains a list of persons. For each person you can record the name, telephone numbers, email adress, and perhaps other relevant data. A natural way of representing such personal data in a program is to create a class, say class `Person`. The attributes of the class holds data like the name, mobile phone number, office phone number, private phone number, and email address. The constructor may initialize some of the data about a person. Additional data can be specified later by calling methods in the class. One method can print the data. Other methods can register additional telephone numbers and an email address. In addition we initialize some of the attributes in a constructor method. The attributes that are not initialized when constructing a `Person` instance can be added later by calling appropriate methods. For example, adding an office number is done by calling `add_office_number`.

Class `Person` may look as

```
class Person:
    def __init__(self, name,
                 mobile_phone=None, office_phone=None,
                 private_phone=None, email=None):
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email

    def add_mobile_phone(self, number):
        self.mobile = number

    def add_office_phone(self, number):
        self.office = number

    def add_private_phone(self, number):
        self.private = number

    def add_email(self, address):
        self.email = address
```

Note the use of `None` as default value for various attributes: the object `None` is commonly used to indicate that a variable or attribute is defined, but yet not with a sensible value.

A quick demo session of class `Person` may go as follows:

```
>>> p1 = Person('Hans Hanson',
...             office_phone='767828283', email='h@hanshanson.com')
>>> p2 = Person('Ole Olsen', office_phone='767828292')
>>> p2.add_email('olsen@somemail.net')
>>> phone_book = [p1, p2]
```

It can be handy to add a method for printing the contents of a `Person` instance in a nice fashion:

```
    def dump(self):
        s = self.name + '\n'
        if self.mobile is not None:
            s += 'mobile phone:   %s\n' % self.mobile
        if self.office is not None:
            s += 'office phone:   %s\n' % self.office
        if self.private is not None:
            s += 'private phone:  %s\n' % self.private
        if self.email is not None:
            s += 'email address:  %s\n' % self.email
        print s
```

With this method we can easily print the phone book:

```
>>> for person in phone_book:
...     person.dump()
...
Hans Hanson
office phone:   767828283
email address:  h@hanshanson.com

Ole Olsen
office phone:   767828292
email address:  olsen@somemail.net
```

A phone book can be a list of `Person` instances, as indicated in the examples above. However, if we quickly want to look up the phone numbers or email address for a given name, it would be more convenient to store the `Person` instances in a dictionary with the name as key:

```
>>> phone_book = {'Hanson': p1, 'Olsen': p2}
>>> for person in sorted(phone_book):  # alphabetic order
...     phone_book[person].dump()
```

The current example of `Person` objects is extended in Chapter 7.3.5.

### 7.2.3 A Circle

Geometric figures, such as a circle, are other candidates for classes in a program. A circle is uniquely defined by its center point $(x_0, y_0)$ and its radius $R$. We can collect these three numbers as attributes in a class. The values of $x_0$, $y_0$, and $R$ are naturally initialized in the constructor. Other methods can be `area` and `circumference` for calculating the area $\pi R^2$ and the circumference $2\pi R$:

```
class Circle:
    def __init__(self, x0, y0, R):
        self.x0, self.y0, self.R = x0, y0, R

    def area(self):
        return pi*self.R**2

    def circumference(self):
        return 2*pi*self.R
```

An example of using class `Circle` goes as follows:

```
>>> c = Circle(2, -1, 5)
>>> print 'A circle with radius %g at (%g, %g) has area %g' % \
...       (c.R, c.x0, c.y0, c.area())
A circle with radius 5 at (2, -1) has area 78.5398
```

The ideas of class `Circle` can be applied to other geometric objects as well: rectangles, triangles, ellipses, boxes, spheres, etc. Exercise 7.4 tests if you are able to adapt class `Circle` to a rectangle and a triangle.

*Remark.* There are usually many solutions to a programming problem. Representing a circle is no exception. Instead of using a class, we could collect $x_0$, $y_0$, and $R$ in a list and create global functions `area` and `circumference` that take such a list as argument:

```
x0, y0, R = 2, -1, 5
circle = [x0, y0, R]

def area(c):
    R = c[2]
    return pi*R**2

def circumference(c):
    R = c[2]
    return 2*pi*R
```

Alternatively, the circle could be represented by a dictionary with keys `'center'` and `'radius'`:

```
circle = {'center': (2, -1), 'radius': 5}

def area(c):
    R = c['radius']
    return pi*R**2

def circumference(c):
    R = c['radius']
    return 2*pi*R
```

## 7.3 Special Methods

Some class methods have names starting and ending with a double underscore. These methods allow a special syntax in the program and are called *special methods*. The constructor `__init__` is one example. This method is automatically called when an instance is created (by calling the class as a function), but we do not need to explicitly write `__init__`. Other special methods make it possible to perform arithmetic operations with instances, to compare instances with `>`, `>=`, `!=`, etc., to call instances as we call ordinary functions, and to test if an instance is true or false, to mention some possibilities.

### 7.3.1 The Call Special Method

Computing the value of the mathematical function represented by class `Y` on page 341, with `y` as the name of the instance, is performed by writing `y.value(t)`. If we could write just `y(t)`, the `y` instance would look as an ordinary function. Such a syntax is indeed possible and offered by the special method named `__call__`. Writing `y(t)` implies a call

```
y.__call__(t)
```

if class `Y` has the method `__call__` defined. We may easily add this special method:

```
class Y:
    ...
    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

The previous `value` method is now redundant. A good programming convention is to include a `__call__` method in all classes that represent a mathematical function. Instances with `__call__` methods are said to be *callable* objects, just as plain functions are callable objects as well. The call syntax for callable objects is the same, regardless of whether the object is a function or a class instance. Given an object `a`, `callable(a)` returns `True` if `a` is either a Python function or an instance with a `__call__` method.

In particular, an instance of class `Y` can be passed as the `f` argument to the `diff` function on page 339:

```
y = Y(v0=5)
dydt = diff(y, 0.1)
```

Inside `diff`, we can test that `f` is not a function but an instance of class `Y`. However, we only use `f` in calls, like `f(x)`, and for this purpose an instance with a `__call__` method works as a plain function. This feature is very convenient.

The next section demonstrates a neat application of the call operator `__call__` in a numerical algorithm.

### 7.3.2 Example: Automagic Differentiation

*Problem.* Given a Python implementation `f(x)` of a mathematical function $f(x)$, we want to create an object that behaves as a Python function for computing the derivative $f'(x)$. For example, if this object is of type `Derivative`, we should be able to write something like

```
>>> def f(x):
        return x**3
...
>>> dfdx = Derivative(f)
>>> x = 2
>>> dfdx(x)
12.000000992884452
```

That is, `dfdx` behaves as a straight Python function for implementing the derivative $3x^2$ of $x^3$ (well, the answer is only approximate, with an error in the 7th decimal, but the approximation can easily be improved).

Maple, Mathematica, and many other software packages can do exact symbolic mathematics, including differentiation and integration. A Python package SymPy for symbolic mathematics is free and simple to use, and could easily be applied to calculate the exact derivative of a large class of functions $f(x)$. However, functions that are defined in an algorithmic way (e.g., solution of another mathematical problem), or functions with branches, random numbers, etc., pose fundamental problems to symbolic differentiation, and then numerical differentiation is required. Therefore we base the computation of derivatives in `Derivative` instances on finite difference formulas. This strategy also leads to much simpler code compared to exact symbolic differentiation.

*Solution.* The most basic, but not the best formula for a numerical derivative is (7.1), which we reuse here for simplicity. The reader can easily switch from this formula to a better one if desired. The idea now is that we make a class to hold the function to be differentiated, call it `f`, and a stepsize `h` to be used in the numerical approximation. These variables can be set in the constructor. The `__call__` operator computes the derivative with aid of the general formula (7.1). All this can be coded as

```
class Derivative:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h      # make short forms
        return (f(x+h) - f(x))/h
```

Note that we turn `h` into a `float` to avoid potential integer division.

Below follows an application of the class to differentiate two functions $f(x) = \sin x$ and $g(t) = t^3$:

```
>>> from math import sin, cos, pi
>>> df = Derivative(sin)
>>> x = pi
>>> df(x)
-1.000000082740371
>>> cos(x)  # exact
-1.0
```

```
>>> def g(t):
...     return t**3
...
>>> dg = Derivative(g)
>>> t = 1
>>> dg(t)  # compare with 3 (exact)
3.000000248221113
```

The expressions `df(x)` and `dg(t)` look as ordinary Python functions that evaluate the derivative of the functions `sin(x)` and `g(t)`. Class `Derivative` works for (almost) any function $f(x)$.

*Application.* In which situations will it be convenient to automatically produce a Python function `df(x)` which is the derivative of another Python function `f(x)`? One example arises when solving nonlinear algebraic equations $f(x) = 0$ with Newton's method and we, because of laziness, lack of time, or lack of training do not manage to derive $f'(x)$ by hand. Consider the `Newton` function from page 248 for solving $f(x) = 0$. Suppose we want to solve

$$f(x) = 10^5(x - 0.9)^2(x - 1.1)^3 = 0$$

by Newton's method. The function $f(x)$ is plotted in Figure 7.2. The

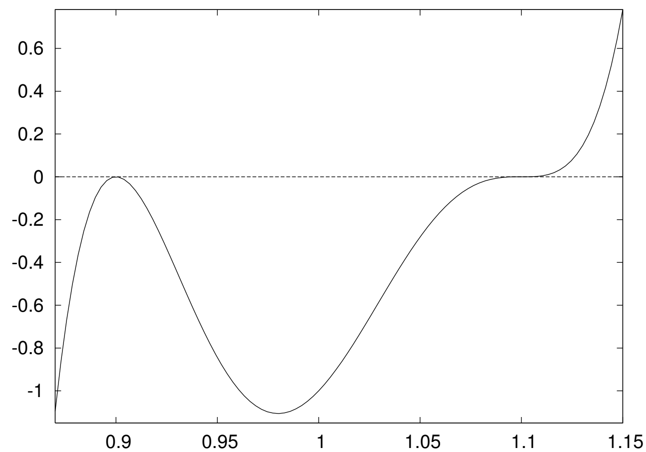



**Fig. 7.2** Plot of $y = 10^5(x - 0.9)^2(x - 1.1)^3$.

following session employs the `Derivative` class to quickly make a derivative so we can call Newton's method:

```
>>> from classes import Derivative
>>> from Newton import Newton
>>> def f(x):
...     return 100000*(x - 0.9)**2 * (x - 1.1)**3
...
>>> df = Derivative(f)
>>> Newton(f, 1.01, df, epsilon=1E-5)
(1.0987610068093443, 8, -7.5139644257961411e-06)
```

The output 3-tuple holds the approximation to a root, the number of iterations, and the value of $f$ at the approximate root (a measure of the error in the equation).

The exact root is 1.1, and the convergence toward this value is very slow[6] (for example, an `epsilon` tolerance of $10^{-10}$ requires 18 iterations with an error of $10^{-3}$). Using an exact derivative gives almost the same result:

```
>>> def df_exact(x):
...     return 100000*(2*(x-0.9)*(x-1.1)**3 + \
...                    (x-0.9)**2*3*(x-1.1)**2)
...
>>> Newton(f, 1.01, df_exact, epsilon=1E-5)
(1.0987610065618421, 8, -7.5139689100699629e-06)
```

This example indicates that there are hardly any drawbacks in using a "smart" inexact general differentiation approach as in the `Derivative` class. The advantages are many – most notably, `Derivative` avoids potential errors from possibly incorrect manual coding of possibly lengthy expressions of possibly wrong hand-calculations. The errors in the involved approximations can be made smaller, usually much smaller than other errors, like the tolerance in Newton's method in this example or the uncertainty in physical parameters in real-life problems.

### 7.3.3 Example: Automagic Integration

We can apply the ideas from Chapter 7.3.2 to make a class for computing the integral of a function numerically. Given a function $f(x)$, we want to compute

$$F(x;a) = \int_a^x f(t)dt\,.$$

The computational technique consists of using the Trapezoidal rule with $n$ intervals ($n+1$ points):

$$\int_a^x f(t)dt = h\left(\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a+ih) + \frac{1}{2}f(x)\right), \qquad (7.2)$$

where $h = (x-a)/n$. In an application program, we want to compute $F(x;a)$ by a simple syntax like

```
def f(x):
    return exp(-x**2)*sin(10*x)
```

[6] Newton's method converges very slowly when the derivative of $f$ is zero at the roots of $f$. Even slower convergence appears when higher-order derivatives also are zero, like in this example. Notice that the error in `x` is much larger than the error in the equation (`epsilon`).

```
a = 0; n = 200
F = Integral(f, a, n)
print F(x)
```

Here, `f(x)` is the Python function to be integrated, and `F(x)` behaves as a Python function that calculates values of $F(x; a)$.

*A Simple Implementation.* Consider a straightforward implementation of the Trapezoidal rule in a Python function:

```
def trapezoidal(f, a, x, n):
    h = (x-a)/float(n)
    I = 0.5*f(a)
    for i in iseq(1, n-1):
        I += f(a + i*h)
    I += 0.5*f(x)
    I *= h
    return I
```

The `iseq` function is an alternative to `range` where the upper limit is included in the set of numbers (see Chapters 4.3.1 and 4.5.6). We can alternatively use `range(1, n)`, but the correspondence with the indices in the mathematical description of the rule is then not completely direct. The `iseq` function is contained in `scitools.std`, so if you make a "star import" from this module, you have `iseq` available.

Class `Integral` must have some attributes and a `__call__` method. Since the latter method is supposed to take `x` as argument, the other parameters `a`, `f`, and `n` must be class attributes. The implementation then becomes

```
class Integral:
    def __init__(self, f, a, n=100):
        self.f, self.a, self.n = f, a, n

    def __call__(self, x):
        return trapezoidal(self.f, self.a, x, self.n)
```

Observe that we just reuse the `trapezoidal` function to perform the calculation. We could alternatively have copied the body of the `trapezoidal` function into the `__call__` method. However, if we already have this algorithm implemented and tested as a function, it is better to call the function. The class is then known as a *wrapper* of the underlying function. A wrapper allows something to be called with alternative syntax. With the `Integral(x)` wrapper we can supply the upper limit of the integral only – the other parameters are supplied when we create an instance of the `Integral` class.

An application program computing $\int_0^{2\pi} \sin x \, dx$ might look as follows:

```
from math import sin, pi

G = Integral(sin, 0, 200)
value = G(2*pi)
```

An equivalent calculation is

```
value = trapezoidal(sin, 0, 2*pi, 200)
```

*Remark.* Class `Integral` is inefficient (but probably more than fast enough) for plotting $F(x;a)$ as a function $x$. Exercise 7.22 suggests to optimize the class for this purpose.

### 7.3.4 Turning an Instance into a String

Another special method is `__str__`. It is called when a class instance needs to be converted to a string. This happens when we say `print a`, and `a` is an instance. Python will then look into the `a` instance for a `__str__` method, which is supposed to return a string. If such a special method is found, the returned string is printed, otherwise just the name of the class is printed. An example will illustrate the point. First we try to print an `y` instance of class `Y` from Chapter 7.1.2 (where there is no `__str__` method):

```
>>> print y
<__main__.Y instance at 0xb751238c>
```

This means that `y` is an `Y` instance in the `__main__` module (the main program or the interactive session). The output also contains an address telling where the `y` instance is stored in the computer's memory.

If we want `print y` to print out the `y` instance, we need to define the `__str__` method in class `Y`:

```
class Y:
    ...
    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Typically, `__str__` replaces our previous `formula` method and `__call__` replaces our previous `value` method. Python programmers with the experience that we now have gained will therefore write class `Y` with special methods only:

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2

    def __str__(self):
        return 'v0*t - 0.5*g*t**2; v0=%g' % self.v0
```

Let us see the class in action:

```
>>> y = Y(1.5)
>>> y(0.2)
0.1038
>>> print y
v0*t - 0.5*g*t**2; v0=1.5
```

What have we gained by using special methods? Well, we can still only evaluate the formula and write it out, but many users of the class will claim that the syntax is more attractive since `y(t)` in code means $y(t)$ in mathematics, and we can do a `print y` to view the formula. The bottom line of using special methods is to achieve a more user-friendly syntax. The next sections illustrate this point further.

### 7.3.5 Example: Phone Book with Special Methods

Let us reconsider class `Person` from Chapter 7.2.2. The `dump` method in that class is better implemented as a `__str__` special method. This is easy: We just change the method name and replace `print s` by `return s`.

Storing `Person` instances in a dictionary to form a phone book is straightforward. However, we make the dictionary a bit easier to use if we wrap a class around it. That is, we make a class `PhoneBook` which holds the dictionary as an attribute. An `add` method can be used to add a new person:

```
class PhoneBook:
    def __init__(self):
        self.contacts = {}   # dict of Person instances

    def add(self, name, mobile=None, office=None,
            private=None, email=None):
        p = Person(name, mobile, office, private, email)
        self.contacts[name] = p
```

A `__str__` can print the phone book in alphabetic order:

```
    def __str__(self):
        s = ''
        for p in sorted(self.contacts):
            s += str(self.contacts[p])
        return s
```

To retrieve a `Person` instance, we use the `__call__` with the person's name as argument:

```
    def __call__(self, name):
        return self.contacts[name]
```

The only advantage of this method is simpler syntax: For a `PhoneBook` `b` we can get data about `NN` by calling `b('NN')` rather than accessing the internal dictionary `b.contacts['NN']`.

We can make a simple test function for a phone book with three names:

```
b = PhoneBook()
b.add('Ole Olsen', office='767828292',
      email='olsen@somemail.net')
b.add('Hans Hanson',
      office='767828283', mobile='995320221')
b.add('Per Person', mobile='906849781')
print b('Per Person')
print b
```

The output becomes

```
Per Person
mobile phone:   906849781

Hans Hanson
mobile phone:   995320221
office phone:   767828283

Ole Olsen
office phone:   767828292
email address:  olsen@somemail.net

Per Person
mobile phone:   906849781
```

You are strongly encouraged to work through this last demo program by hand and simulate what the program does. That is, jump around in the code and write down on a piece of paper what various variables contain after each statement. This is an important and good exercise! You enjoy the happiness of mastering classes if you get the same output as above. The complete program with classes `Person` and `PhoneBook` and the test above is found in the file `phone_book.py`. You can run this program, statement by statement, in a debugger (see Appendix D.1) to control that your understanding of the program flow is correct.

*Remark.* Note that the names are sorted with respect to the first names. The reason is that strings are sorted after the first character, then the second character, and so on. We can supply our own tailored sort function, as explained in Exercise 2.44. One possibility is to split the name into words and use the last word for sorting:

```
def last_name_sort(name1, name2):
    lastname1 = name1.split()[-1]
    lastname2 = name2.split()[-1]
    if lastname1 < lastname2:
        return -1
    elif lastname1 > lastname2:
        return 1
    else: # equality
        return 0

for p in sorted(self.contacts, last_name_sort):
    ...
```

### 7.3.6 Adding Objects

Let `a` and `b` be instances of some class `C`. Does it make sense to write `a + b`? Yes, this makes sense if class `C` has defined a special method `__add__`:

```
class C:
    ...
    __add__(self, other):
        ...
```

The `__add__` method should add the instances `self` and `other` and return the result as an instance. So when Python encounters `a + b`, it will check if class `C` has an `__add__` method and interpret `a + b` as the call `a.__add__(b)`. The next example will hopefully clarify what this idea can be used for.

### 7.3.7 Example: Class for Polynomials

Let us create a class `Polynomial` for polynomials. The coefficients in the polynomial can be given to the constructor as a list. Index number $i$ in this list represents the coefficients of the $x^i$ term in the polynomial. That is, writing `Polynomial([1,0,-1,2])` defines a polynomial

$$1 + 0 \cdot x - 1 \cdot x^2 + 2 \cdot x^3 = 1 - x^2 + 2x^3 \, .$$

Polynomials can be added (by just adding the coefficients) so our class may have an `__add__` method. A `__call__` method is natural for evaluating the polynomial, given a value of $x$. The class is listed below and explained afterwards.

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        # start with the longest list and add in the other:
        if len(self.coeff) > len(other.coeff):
            sum_coeff = self.coeff[:]  # copy!
            for i in range(len(other.coeff)):
                sum_coeff[i] += other.coeff[i]
        else:
            sum_coeff = other.coeff[:] # copy!
            for i in range(len(self.coeff)):
                sum_coeff[i] += self.coeff[i]
        return Polynomial(sum_coeff)
```

*Implementation.* Class `Polynomial` has one attribute: the list of coefficients. To evaluate the polynomial, we just sum up coefficient no. $i$ times $x^i$ for $i = 0$ to the number of coefficients in the list.

The `__add__` method looks more advanced. The idea is to add the two lists of coefficients. However, it may happen that the lists are of unequal length. We therefore start with the longest list and add in the other list, element by element. Observe that `sum_coeff` starts out as a *copy* of `self.coeff`: If not, changes in `sum_coeff` as we compute the sum will be reflected in `self.coeff`. This means that `self` would be the sum of itself and the `other` instance, or in other words, adding two instances, `p1+p2`, changes `p1` – this is not what we want! An alternative implementation of class `Polynomial` is found in Exercise 7.32.

A subtraction method `__sub__` can be implemented along the lines of `__add__`, but is slightly more complicated and left to the reader through Exercise 7.33. A somewhat more complicated operation, from a mathematical point of view, is the multiplication of two polynomials. Let $p(x) = \sum_{i=0}^{M} c_i x^i$ and $q(x) = \sum_{j=0}^{N} d_j x^j$ be the two polynomials. The product becomes

$$\left(\sum_{i=0}^{M} c_i x^i\right) \left(\sum_{j=0}^{N} d_j x^j\right) = \sum_{i=0}^{M} \sum_{j=0}^{N} c_i d_j x^{i+j} \, .$$

The double sum must be implemented as a double loop, but first the list for the resulting polynomial must be created with length $M+N+1$ (the highest exponent is $M+N$ and then we need a constant term). The implementation of the multiplication operator becomes

```
    def __mul__(self, other):
        c = self.coeff
        d = other.coeff
        M = len(c) - 1
        N = len(d) - 1
        result_coeff = zeros(M+N-1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                result_coeff[i+j] += c[i]*d[j]
        return Polynomial(result_coeff)
```

We could also include a method for differentiating the polynomial according to the formula

$$\frac{d}{dx} \sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1} \, .$$

If $c_i$ is stored as a list `c`, the list representation of the derivative, say its name is `dc`, fulfills `dc[i-1] = i*c[i]` for `i` running from 1 to the largest index in `c`. Note that `dc` has one element less than `c`.

There are two different ways of implementing the differentiation functionality, either by changing the polynomial coefficients, or by re-

turning a new `Polynomial` instance from the method such that the original polynomial instance is intact. We let `p.differentiate()` be an implementation of the first approach, i.e., this method does not return anything, but the coefficients in the `Polynomial` instance `p` are altered. The other approach is implemented by `p.derivative()`, which returns a new `Polynomial` object with coefficients corresponding to the derivative of `p`.

The complete implementation of the two methods is given below:

```
    def differentiate(self):
        """Differentiate this polynomial in-place."""
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self):
        """Copy this polynomial and return its derivative."""
        dpdx = Polynomial(self.coeff[:])  # make a copy
        dpdx.differentiate()
        return dpdx
```

The `Polynomial` class with a `differentiate` method and not a `derivative` method would be mutable (see Chapter 6.2.3) and allow in-place changes of the data, while the `Polynomial` class with `derivative` and not `differentiate` would yield an immutable object where the polynomial initialized in the constructor is never altered[7]. A good rule is to offer only one of these two functions such that a `Polynomial` object is either mutable or immutable (if we leave out `differentiate`, its function body must of course be copied into `derivative` since `derivative` now relies on that code). However, since the main purpose of this class is to illustrate various types of programming techniques, we keep both versions.

*Usage.* As a demonstration of the functionality of class `Polynomial`, we introduce the two polynomials

$$p_1(x) = 1 - x, \quad p_2(x) = x - 6x^4 - x^5 \,.$$

```
>>> p1 = Polynomial([1, -1])
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = p1 + p2
>>> print p3.coeff
[1, 0, 0, 0, -6, -1]
>>> p4 = p1*p2
>>> print p4.coeff
[0, 1, -1, 0, -6, 5, 1]
>>> p5 = p2.derivative()
```

[7] Technically, it is possible to grab the `coeff` variable in a class instance and alter this list. By starting `coeff` with an underscore, a Python programming convention tells programmers that this variable is for internal use in the class only, and not to be altered by users of the instance, see Chapters 7.2.1 and 7.6.2.

```
>>> print p5.coeff
[1, 0, 0, -24, -5]
```

One verification of the implementation may be to compare `p3` at (e.g.) $x = 1/2$ with $p_1(x) + p_2(x)$:

```
>>> x = 0.5
>>> p1_plus_p2_value = p1(x) + p2(x)
>>> p3_value = p3(x)
>>> print p1_plus_p2_value - p3_value
0.0
```

Note that `p1 + p2` is very different from `p1(x) + p2(x)`. In the former case, we add two instances of class `Polynomial`, while in the latter case we add two instances of class `float` (since `p1(x)` and `p2(x)` imply calling `__call__` and that method returns a `float` object).

*Pretty Print of Polynomials.* The `Polynomial` class can also be equipped with a `__str__` method for printing the polynomial to the screen. A first, rough implementation could simply add up strings of the form `+ self.coeff[i]*x^i`:

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(len(self.coeff)):
            s += ' + %g*x^%d' % (self.coeff[i], i)
        return s
```

However, this implementation leads to ugly output from a mathematical viewpoint. For instance, a polynomial with coefficients `[1,0,0,-1,-6]` gets printed as

```
 + 1*x^0 + 0*x^1 + 0*x^2 + -1*x^3 + -6*x^4
```

A more desired output would be

```
1 - x^3 - 6*x^4
```

That is, terms with a zero coefficient should be dropped; a part `'+ -'` of the output string should be replaced by `'- '`; unit coefficients should be dropped, i.e., `' 1*'` should be replaced by space `' '`; unit power should be dropped by replacing `'x^1 '` by `'x '`; zero power should be dropped and replaced by `1`, initial spaces should be fixed, etc. These adjustments can be implemented using the `replace` method in string objects and by composing slices of the strings. The new version of the `__str__` method below contains the necessary adjustments. If you find this type of string manipulation tricky and difficult to understand, you may safely skip further inspection of the improved `__str__` code since the details are not essential for your present learning about the class concept and special methods.

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += ' + %g*x^%d' % (self.coeff[i], i)
        # fix layout:
        s = s.replace('+ -', '- ')
        s = s.replace('x^0', '1')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^1 ', 'x ')
        s = s.replace('x^1', 'x')
        if s[0:3] == ' + ':  # remove initial +
            s = s[3:]
        if s[0:3] == ' - ':  # fix spaces for initial -
            s = '-' + s[3:]
        return s
```

Programming sometimes turns into coding (what one think is) a general solution followed by a series of special cases to fix caveats in the "general" solution, just as we experienced with the `__str__` method above. This situation often calls for additional future fixes and is often a sign of a suboptimal solution to the programming problem.

Pretty print of `Polynomial` instances can be demonstrated in an interactive session:

```
>>> p1 = Polynomial([1, -1])
>>> print p1
1 - x^1
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p2.differentiate()
>>> print p2
1 - 24*x^3 - 5*x^4
```

### 7.3.8 Arithmetic Operations and Other Special Methods

Given two instances `a` and `b`, the standard binary arithmetic operations with `a` and `b` are defined by the following special methods:

- `a + b` : `a.__add__(b)`
- `a - b` : `a.__sub__(b)`
- `a*b` : `a.__mul__(b)`
- `a/b` : `a.__div__(b)`
- `a**b` : `a.__pow__(b)`

Some other special methods are also often useful:

- the length of `a`, `len(a)`: `a.__len__()`
- the absolute value of `a`, `abs(a)`: `a.__abs__()`
- `a == b` : `a.__eq__(b)`
- `a > b` : `a.__gt__(b)`
- `a >= b` : `a.__ge__(b)`
- `a < b` : `a.__lt__(b)`

- `a <= b` : `a.__le__(b)`
- `a != b` : `a.__ne__(b)`
- `-a` : `a.__neg__()`
- evaluating `a` as a boolean expression (as in the test `if a:`) implies calling the special method `a.__bool__()`, which must return `True` or `False` – if `__bool__` is not defined, `__len__` is called to see if the length is zero (`False`) or not (`True`)

We can implement such methods in class `Polynomial`, see Exercise 7.33. Chapter 7.5 contains many examples on using the special methods listed above.

### 7.3.9 More on Special Methods for String Conversion

Look at this class with a `__str__` method:

```
>>> class MyClass:
...     def __init__(self):
...         self.data = 2
...     def __str__(self):
...         return 'In __str__: %s' % str(self.data)
...
>>> a = MyClass()
>>> print a
In __str__: 2
```

Hopefully, you understand well why we get this output (if not, go back to Chapter 7.3.4).

But what will happen if we write just `a` at the command prompt in an interactive shell?

```
>>> a
<__main__.MyClass instance at 0xb75125ac>
```

When writing just `a` in an interactive session, Python looks for a special method `__repr__` in `a`. This method is similar to `__str__` in that it turns the instance into a string, but there is a convention that `__str__` is a pretty print of the instance contents while `__repr__` is a complete represention of the contents of the instance. For a lot of Python classes, including `int`, `float`, `complex`, `list`, `tuple`, and `dict`, `__repr__` and `__str__` give identical output. In our class `MyClass` the `__repr__` is missing, and we need to add it if we want

```
>>> a
```

to write the contents like `print a` does.

Given an instance `a`, `str(a)` implies calling `a.__str__()` and `repr(a)` implies calling `a.__repr__()`. This means that

```
>>> a
```

is actually a `repr(a)` call and

```
>>> print a
```

is actually a `print str(a)` statement.

A simple remedy in class `MyClass` is to define

```
def __repr__(self):
    return self.__str__()  # or return str(self)
```

However, as we explain below, the `__repr__` is best defined differently.

*Recreating Objects from Strings.* The Python function `eval(e)` evaluates a valid Python expression contained in the string `e`, see Chapter 3.1.2. It is a convention that `__repr__` returns a string such that `eval` applied to the string recreates the instance. For example, in case of the `Y` class from page 341, `__repr__` should return `'Y(10)'` if the `v0` variable has the value `10`. Then `eval('Y(10)')` will be the same as if we had coded `Y(10)` directly in the program or an interactive session.

Below we show examples of `__repr__` methods in classes `Y` (page 341), `Polynomial` (page 365), and `MyClass` (above):

```
class Y:
    ...
    def __repr__(self):
        return 'Y(v0=%s)' % self.v0

class Polynomial:
    ...
    def __repr__(self):
        return 'Polynomial(coefficients=%s)' % self.coeff

class MyClass:
    ...
    def __repr__(self):
        return 'MyClass()'
```

With these definitions, `eval(repr(x))` recreates the object `x` if it is of one of the three types above. In particular, we can write `x` to file and later recreate the `x` from the file information:

```
# somefile is some file object
somefile.write(repr(x))
somefile.close()
...
data = somefile.readline()
x2 = eval(data)  # recreate object
```

Now, `x2` will be equal to `x` (`x2 == x` evaluates to true).

## 7.4 Example: Solution of Differential Equations

An ordinary differential equation (ODE), where the unknown is a function $u(t)$, can be written in the generic form

$$u'(t) = f(u(t), t)\,. \tag{7.3}$$

In addition, an initial condition, $u(0) = u_0$, must be associated with this ODE to make the solution of (7.3) unique. The function $f$ reflects an expression with $u$ and/or $t$. Some important examples of ODEs and their corresponding forms of $f$ are given below.

1. Exponential growth of money or populations:
$$f(u, t) = \alpha u, \tag{7.4}$$
where $\alpha$ is a given constant expressing the growth rate of $u$.
2. Logistic growth of a population under limited resources:
$$f(u, t) = \alpha u \left(1 - \frac{u}{R}\right), \tag{7.5}$$
where $\alpha$ is the initial growth rate and $R$ is the maximum possible value of $u$.
3. Radioactive decay of a substance:
$$f(u, t) = -au, \tag{7.6}$$
where $a$ is the rate of decay of $u$.
4. Body falling in a fluid:
$$f(u, t) = -b|u|u + g, \tag{7.7}$$
where $b$ models the fluid resistance, $g$ is the acceleration of gravity, and $u$ is the body's velocity (see Exercise 7.25 on page 405).
5. Newton's law of cooling:
$$f(u, t) = -h(u - s), \tag{7.8}$$
where $u$ is the temperature of a body, $h$ is a heat transfer coefficient between the body and its surroundings, and $s$ is the temperature of the surroundings.

Appendix B gives an introduction to ODEs and their numerical solution, and you should be familiar with that or similar material before reading on.

The purpose of the present section is to design and implement a class for the general ODE $u' = f(u, t)$. You need to have digested the material about classes in Chapters 7.1.2, 7.3.1, and 7.3.2.

### 7.4.1 A Function for Solving ODEs

A very simple solution method for a general ODE on the form (7.3) is the Forward Euler method:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)\,. \tag{7.9}$$

Here, $u_k$ denotes the numerical approximation to the exact solution $u$ at time $t_k$, $\Delta t$ is a time step, and if all time steps are equal, we have that $t_k = k\Delta t$, $k = 0, \ldots, n$.

First we will implement the method (7.9) in a simple program, tailored to the specific ODE $u' = u$ (i.e., $f(u,t) = u$ in (7.3)). Our goal is to compute $u(t)$ for $t \in [0, T]$. How to perform this computation is explained in detail in Appendix B.2. Here, we follow the same approach, but using lists instead of arrays to store the computed $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$ values. The reason is that lists are more dynamical if we later introduce more sophisticated solution methods where $\Delta t$ may change during the solution so that the value of $n$ (i.e., length of arrays) is not known on beforehand. Let us start with sketching a "flat program":

```
# Integrate u'=u, u(0)=u0, in steps of dt until t=T
u0 = 1
T = 3
dt = 0.1

u = []; t = []   # u[k] is the solution at time t[k]

u.append(u0)
t.append(0)
n = int(round(T/dt))
for k in range(n):
    unew = u[k] + dt*u[k]

    u.append(unew)
    tnew = t[-1] + dt
    t.append(tnew)
from scitools.std import plot
plot(t, u)
```

The new $u_{k+1}$ and $t_{k+1}$ values, stored in the `unew` and `tnew` variables, are appended to the `u` and `t` lists in each pass in the loop over `k`.

Unfortunately, the code above can only be applied to a specific ODE using a specific numerical method. An obvious improvement is to make a *reusable function* for solving a *general ODE*. An ODE is specified by its right-hand side function $f(u, t)$. We also need the parameters $u_0$, $\Delta t$, and $T$ to perform the time stepping. The function should return the computed $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$ values as two separate arrays to the calling code. These arrays can then be used for plotting or data analysis. An appropriate function may look like

```
def ForwardEuler(f, dt, u0, T):
    """Integrate u'=f(u,t), u(0)=u0, in steps of dt until t=T."""
    u = []; t = []   # u[k] is the solution at time t[k]
    u.append(u0)
```

```
    t.append(0)
    n = int(round(T/dt))
    for k in range(n):
        unew = u[k] + dt*f(u[k], t[k])

        u.append(unew)
        tnew = t[-1] + dt
        t.append(tnew)
    return numpy.array(u), numpy.array(t)
```

Here, `f(u, t)` is a Python implementation of $f(u, t)$ that the user must supply. For example, we may solve $u' = u$ for $t \in (0, 3)$, with $u(0) = 1$, and $\Delta t = 0.1$ by the following code utilizing the shown `ForwardEuler` function:

```
def f(u, t):
    return u

u0 = 1
T = 3
dt = 0.1
u, t = ForwardEuler(f, dt, u0, T)

# compare numerical solution and exact solution in a plot:
from scitools.std import plot, exp
u_exact = exp(t)
plot(t, u, 'r-', t, u_exact, 'b-',
     xlabel='t', ylabel='u', legend=('numerical', 'exact'),
     title="Solution of the ODE u'=u, u(0)=1")
```

Observe how easy it is to plot `u` versus `t` and also add the exact solution $u = e^t$ for comparison.

### 7.4.2 A Class for Solving ODEs

Instead of having the numerical method for solving a general ODE implemented as a function, we now want a class for this purpose. Say the name of the class is `ForwardEuler`. To solve an ODE specified by a Python function `f(u, t)`, from time `t0` to some time `T`, with steps of size `dt`, and initial condition `u0` at time `t0`, it seems convenient to write the following lines of code:

```
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, t0)
u, t = method.solve(T)
```

The constructor of the class stores `f` and the time step `dt`. Then there are two basic steps: setting the initial condition, and calling `solve` to advance the solution to a specified time level. Observe that we do not force the initial condition to appear at $t = 0$, but at some arbitrary time. This makes the code more general, and in particular, we can call the `solve` again to advance the solution further in time, say to $2T$:

```
method.set_initial_condition(u[-1], t[-1])
u2, t2 = method.solve(2*T)
plot(t, u, 'r-', t2, u2, 'r-')
```

The initial condition of this second simulation is the final `u` and `t` values of the first simulation. To plot the complete solution, we just plot the individual simulations.

The task now is to write a class `ForwardEuler` that allow this type of user code. Much of the code from the `ForwardEuler` function above can be reused, but it is reorganized into smaller pieces in a class. Such reorganization is known as refactoring (see also Chapter 3.6.2). An attempt to write the class appears below.

```
class ForwardEuler:
    """
    Class for solving an ODE,

      du/dt = f(u, t)

    by the ForwardEuler method.

    Class attributes:
    t: array of time values
    u: array of solution values (at time points t)
    k: step number of the most recently computed solution
    f: callable object implementing f(u, t)
    dt: time step (assumed constant)
    """
    def __init__(self, f, dt):
        self.f, self.dt = f, dt

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process

        self.u.append(float(u0))
        self.t.append(float(t0))
        self.k = 0  # time level counter

    def solve(self, T):
        """Advance solution in time until t <= T."""
        tnew = 0
        while tnew <= T:
            unew = self.advance()
            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)

    def advance(self):
        """Advance the solution one time step."""
        # load attributes into local variables to
        # obtain a formula that is as close as possible
        # to the mathematical notation:
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        unew = u[k] + dt*f(u[k], t)
        return unew
```

We see that we initialize two lists for the $u_k$ and $t_k$ values at the time we set the initial condition. The `solve` method implements the time loop as in the `ForwardEuler` function. However, inside the time loop, a new $u_{k+1}$ value (`unew`) is computed by calling another class method, `self.advance`, which here implements the numerical method (7.9).

Changing the numerical method is just a matter of changing the `advance` function only. We could, of course, put the numerical updating formula explicitly inside the `solve` method, but changing the numerical method would then imply editing internals of a method rather than replacing a complete method. The latter task is considered less error-prone and therefore a better programming strategy.

### 7.4.3 Verifying the Implementation

We need a problem where the exact solution is known to check the correctness of class `ForwardEuler`. Preferably, we should have a problem where the numerical solution is exact such that we avoid dealing with approximation errors in the Forward Euler method. It turns out that if the solution $u(t)$ is linear in $t$, then the Forward Euler method will reproduce this solution exactly. Therefore, we choose $u(t) = at + u_0$, with (e.g.) $a = 0.2$ and $u_0 = 3$. The corresponding $f$ is the derivative of $u$, i.e., $f(u, t) = a$. This is obviously a very simple right-hand side without any $u$ or $t$. We can make $f$ more complicated by adding something that is zero, e.g., some expression with $u - at - u_0$, say $(u - at - u_0)^4$, so that $f(u, t) = a + (u - at - u_0)^4$.

We implement our special $f$ and the exact solution in two functions `_f1` and `_u_solution_f1`:

```
def _f1(u, t):
    return 0.2 + (u - _u_solution_f1(t))**4

def _u_solution_f1(t):
    return 0.2*t + 3
```

Testing the code is now a matter of performing the steps

```
u0 = 3
dt = 0.4
T = 3
method = ForwardEuler(_f1, dt)
method.set_initial_condition(u0, 0)
u, t = method.solve(T)
u_exact = _f1_solution(t)
print 'Numerical:\n', u
print 'Exact:', '\n', u_exact
```

The output becomes

```
Numerical:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56  3.64]
Exact:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56  3.64]
```

showing that the code works as it should in this example.

### 7.4.4 Example: Logistic Growth

A more exciting application is to solve the logistic equation (B.23),

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right),$$

with the $f(u,t)$ function specified in (7.5).

First we may create a class for holding information about this problem: $\alpha$, $R$, the initial condition, and the right-hand side. We may also add a method for printing the equation and initial condition. This problem class can then be expressed as

```
class Logistic:
    """Problem class for a logistic ODE."""
    def __init__(self, alpha, R, u0):
        self.alpha, self.R, self.u0 = alpha, float(R), u0

    def __call__(self, u, t):
        """Return f(u,t) for the logistic ODE."""
        return self.alpha*u*(1 - u/self.R)

    def __str__(self):
        """Return ODE and initial condition."""
        return "u'(t) = %g*u*(1 - u/%g)\nu(0)=%g" % \
               (self.alpha, self.R, self.u0)
```

Running a case with $\alpha = 0.2$, $R = 1$, $u(0) = 0.1$, $\Delta t = 0.1$, and simulating up to time $T = 40$, can be performed in the following function:

```
def logistic():
    problem = Logistic(0.2, 1, 0.1)
    T = 40
    dt = 0.1
    method = ForwardEuler(problem, dt)
    method.set_initial_condition(problem.u0, 0)
    u, t = method.solve(T)

    from scitools.std import plot, hardcopy, xlabel, ylabel, title
    plot(t, u)
    xlabel('t'); ylabel('u')
    title('Logistic growth: alpha=0.2, dt=%g, %d steps' \
          % (dt, len(u)-1))
```

The resulting plot is shown in Figure 7.3. Note one aspect of this function: the "star import", as in `from scitools.std import *`, is not allowed *inside* a function (or class method for that sake), so we need to explicitly list all the functions we need to import. (We could, as in the previous example, just import `plot` and rely on keyword arguments to set the labels, title, and output file.)

The `ForwardEuler` class is further developed in Chapter 9.4, where it is shown how we can easily modify the class to implement other numerical methods. In that chapter we extend the implementation to systems of ODEs as well.

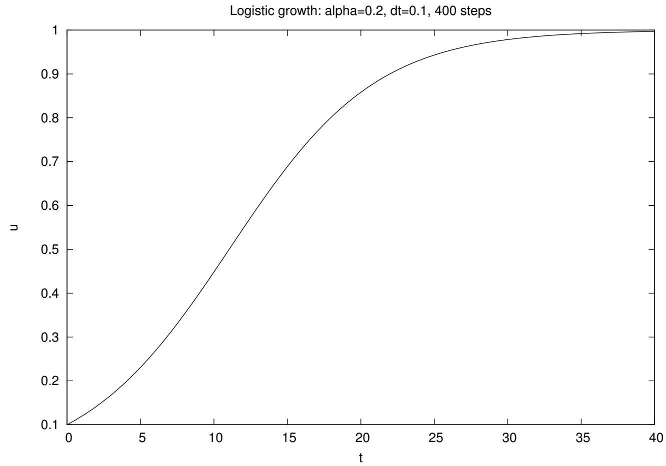



**Fig. 7.3** Plot of the solution of the ODE problem $u' = 0.2u(1-u)$, $u(0) = 0.1$.

## 7.5 Example: Class for Vectors in the Plane

This section explains how to implement two-dimensional vectors in Python such that these vectors act as objects we can add, subtract, form inner products with, and do other mathematical operations on. To understand the forthcoming material, it is necessary to have digested Chapter 7.3, in particular Chapters 7.3.6 and 7.3.8.

### 7.5.1 Some Mathematical Operations on Vectors

Vectors in the plane are described by a pair of real numbers, $(a, b)$. In Chapter 4.1.2 we presented mathematical rules for adding and subtracting vectors, multiplying two vectors (the inner or dot or scalar product), the length of a vector, and multiplication by a scalar:

$$(a, b) + (c, d) = (a + c, b + d), \tag{7.10}$$

$$(a, b) - (c, d) = (a - c, b - d), \tag{7.11}$$

$$(a, b) \cdot (c, d) = ac + bd, \tag{7.12}$$

$$||(a, b)|| = \sqrt{(a, b) \cdot (a, b)}\,. \tag{7.13}$$

Moreover, two vectors $(a, b)$ and $(c, d)$ are equal if $a = c$ and $b = d$.

### 7.5.2 Implementation

We may create a class for plane vectors where the above mathematical operations are implemented by special methods. The class must contain two attributes, one for each component of the vector, called `x` and `y` below. We include special methods for addition, subtraction, the scalar

product (multiplication), the absolute value (length), comparison of two vectors (`==` and `!=`), as well as a method for printing out a vector.

```
class Vec2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vec2D(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vec2D(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y

    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)

    def __ne__(self, other):
        return not self.__eq__(other)  # reuse __eq__
```

The `__add__`, `__sub__`, `__mul__`, `__abs__`, and `__eq__` methods should be quite straightforward to understand from the previous mathematical definitions of these operations. The last method deserves a comment: Here we simply reuse the equality operator `__eq__`, but preceed it with a `not`. We could also have implemented this method as

```
    def __ne__(self, other):
        return self.x != other.x or self.y != other.y
```

Nevertheless, this implementation requires us to write more, and it has the danger of introducing an error in the logics of the boolean expressions. A more reliable approach, when we know that the `__eq__` method works, is to reuse this method and observe that "`not ==`" gives us the effect of "`!=`".

A word of warning is in place regarding our implementation of the equality operator (`==` via `__eq__`). We test for equality of each component, which is correct from a mathematical point of view. However, each vector component is a floating-point number that may be subject to round-off errors both in the representation on the computer and from previous (inexact) floating-point calculations. Two mathematically equal components may be different in their inexact representations on the computer. The remedy for this problem is to avoid testing for equality, but instead check that the difference between the components is sufficiently small. The function `float_eq` found in the module `scitools.numpytutils` (if you do not already have `float_eq` from a `from`

`scitools.std import *`), see also Exercise 2.51, is an easy-to-use tool for comparing `float` objects. With this function we replace

```
if a == b:
```

by

```
if float_eq(a, b):
```

A more reliable equality operator can now be implemented:

```
class Vec2D:
    ...
    def __eq__(self, other):
        return float_eq(self.x, other.x) and \
               float_eq(self.y, other.y)
```

As a rule of thumb, you should never apply the `==` test to two `float` objects.

The special method `__len__` could be introduced as a synonym for `__abs__`, i.e., for a `Vec2D` instance named `v`, `len(v)` is the same as `abs(v)`, because the absolute value of a vector is mathematically the same as the length of the vector. However, if we implement

```
    def __len__(self):
        # reuse implementation of __abs__
        return abs(self)  # equiv. to self.__abs__()
```

we will run into trouble when we compute `len(v)` and the answer is (as usual) a `float`. Python will then complain and tell us that `len(v)` must return an `int`. Therefore, `__len__` cannot be used as a synonym for the length of the vector in our application. On the other hand, we could let `len(v)` mean the number of components in the vector:

```
    def __len__(self):
        return 2
```

This is not a very useful function, though, as we already know that all our `Vec2D` vectors have just two components. For generalizations of the class to vectors with $n$ components, the `__len__` method is of course useful.

### 7.5.3 Usage

Let us play with some `Vec2D` objects:

```
>>> u = Vec2D(0,1)
>>> v = Vec2D(1,0)
>>> w = Vec2D(1,1)
>>> a = u + v
>>> print a
(1, 1)
```

```
>>> a == w
True
>>> a = u - v
>>> print a
(-1, 1)
>>> a = u*v
>>> print a
0
>>> print abs(u)
1.0
>>> u == v
False
>>> u != v
True
```

When you read through this interactive session, you should check that the calculation is mathematically correct, that the resulting object type of a calculation is correct, and how each calculation is performed in the program. The latter topic is investigated by following the program flow through the class methods. As an example, let us consider the expression `u != v`. This is a boolean expression that is true since `u` and `v` are different vectors. The resulting object type should be `bool`, with values `True` or `False`. This is confirmed by the output in the interactive session above. The Python calculation of `u != v` leads to a call to

```
u.__ne__(v)
```

which leads to a call to

```
u.__eq__(v)
```

The result of this last call is `False`, because the special method will evaluate the boolean expression

```
0 == 1 and 1 == 0
```

which is obviously `False`. When going back to the `__ne__` method, we end up with a return of `not False`, which evaluates to `True`. You need this type of thorough understanding to find and correct bugs (and remember that the first versions of your programs will normally contain bugs!).

*Comment.* For real computations with vectors in the plane, you would probably just use a Numerical Python array of length 2. However, one thing such objects cannot do is evaluating `u*v` as a scalar product. The multiplication operator for Numerical Python arrays is not defined as a scalar product (it is rather defined as $(a, b) \cdot (c, d) = (ac, bd)$). Another difference between our `Vec2D` class and Numerical Python arrays is the `abs` function, which computes the length of the vector in class `Vec2D`, while it does something completely different with Numerical Python arrays.

## 7.6 Example: Class for Complex Numbers

Imagine that Python did not already have complex numbers. We could then make a class for such numbers and support the standard mathematical operations. This exercise turns out to be a very good pedagogical example of programming with classes and special methods.

The class must contain two attributes: the real and imaginary part of the complex number. In addition, we would like to add, subtract, multiply, and divide complex numbers. We would also like to write out a complex number in some suitable format. A session involving our own complex numbers may take the form

```
>>> u = Complex(2,-1)
>>> v = Complex(1)     # zero imaginary part
>>> w = u + v
>>> print w
(3, -1)
>>> w != u
True
>>> u*v
Complex(2, -1)
>>> u < v
illegal operation "<" for complex numbers
>>> print w + 4
(7, -1)
>>> print 4 - w
(1, 1)
```

We do not manage to use exactly the same syntax with `j` as imaginary unit as in Python's built-in complex numbers so to specify a complex number we must create a `Complex` instance.

### 7.6.1 Implementation

Here is the complete implementation of our class for complex numbers:

```
class Complex:
    def __init__(self, real, imag=0.0):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return Complex(self.real + other.real,
                       self.imag + other.imag)

    def __sub__(self, other):
        return Complex(self.real - other.real,
                       self.imag - other.imag)

    def __mul__(self, other):
        return Complex(self.real*other.real - self.imag*other.imag,
                       self.imag*other.real + self.real*other.imag)

    def __div__(self, other):
        sr, si, or, oi = self.real, self.imag, \
                         other.real, other.imag # short forms
        r = float(or**2 + oi**2)
```

```
        return Complex((sr*or+si*oi)/r, (si*or-sr*oi)/r)

    def __abs__(self):
        return sqrt(self.real**2 + self.imag**2)

    def __neg__(self):   # defines -c (c is Complex)
        return Complex(-self.real, -self.imag)

    def __eq__(self, other):
        return self.real == other.real and self.imag == other.imag

    def __ne__(self, other):
        return not self.__eq__(other)

    def __str__(self):
        return '(%g, %g)' % (self.real, self.imag)

    def __repr__(self):
        return 'Complex' + str(self)

    def __pow__(self, power):
        raise NotImplementedError\
              ('self**power is not yet impl. for Complex')
```

The special methods for addition, subtraction, multiplication, division, and the absolute value follow easily from the mathematical definitions of these operations for complex numbers (see Chapter 1.6). What `-c` means when `c` is of type `Complex`, is also easy to define and implement. The `__eq__` method needs a word of caution: The method is mathematically correct, but as we stated on page 379, comparison of real numbers on a computer should always employ a tolerance. The version of `__eq__` shown above is more about compact code and equivalence to the mathematics than real-world numerical computations.

The final `__pow__` method exemplifies a way to introduce a method in a class, while we postpone its implementation. The simplest way to do this is by inserting an empty function body using the `pass` ("do nothing") statement:

```
    def __pow__(self, power):
        # postpone implementation of self**power
        pass
```

However, the preferred method is to raise a `NotImplementedError` exception so that users writing power expressions are notified that this operation is not available. The simple `pass` will just silently bypass this serious fact!

### 7.6.2 Illegal Operations

Some mathematical operations, like the comparison operators `>`, `>=`, etc., do not have a meaning for complex numbers. By default, Python allows us to use these comparison operators for our `Complex` instances, but the boolean result will be mathematical nonsense. Therefore, we

should implement the corresponding special methods and give a sensible error message that the operations are not available for complex numbers. Since the messages are quite similar, we make a separate method to gather common operations:

```
    def _illegal(self, op):
        print 'illegal operation "%s" for complex numbers' % op
```

Note the underscore prefix: This is a Python convention telling that the `_illegal` method is local to the class in the sense that it is not supposed to be used outside the class, just by other class methods. In computer science terms, we say that names starting with an underscore are not part of the *application programming interface*, known as the API. Other programming languages, such as Java, C++, and C#, have special keywords, like `private` and `protected` that can be used to technically hide both data and methods from users of the class. Python will never restrict anybody who tries to access data or methods that are considered private to the class, but the leading underscore in the name reminds any user of the class that she now touches parts of the class that are not meant to be used "from the outside".

Various special methods for comparison operators can now call up `_illegal` to issue the error message:

```
    def __gt__(self, other):  self._illegal('>')
    def __ge__(self, other):  self._illegal('>=')
    def __lt__(self, other):  self._illegal('<')
    def __le__(self, other):  self._illegal('<=')
```

### 7.6.3 Mixing Complex and Real Numbers

The implementation of class `Complex` is far from perfect. Suppose we add a complex number and a real number, which is a mathematically perfectly valid operation:

```
w = u + 4.5
```

This statement leads to an exception,

```
AttributeError: 'float' object has no attribute 'real'
```

In this case, Python sees `u + 4.5` and tries to use `u.__add__(4.5)`, which causes trouble because the `other` argument in the `__add__` method is 4.5, i.e., a `float` object, and `float` objects do not contain an attribute with the name `real` (`other.real` is used in our `__add__` method, and accessing `other.real` is what causes the error).

One idea for a remedy could be to set

```
other = Complex(other)
```

since this construction turns a real number `other` into a `Complex` object. However, when we add two `Complex` instances, `other` is of type `Complex`, and the constructor simply stores this `Complex` instance as `self.real` (look at the method `__init__`). This is not what we want!

A better idea is to test for the type of `other` and perform the right conversion to `Complex`:

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            other = Complex(other)
        return Complex(self.real + other.real,
                       self.imag + other.imag)
```

We could alternatively drop the conversion of `other` and instead implement two addition rules, depending on the type of `other`:

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            return Complex(self.real + other, self.imag)
        else:
            return Complex(self.real + other.real,
                           self.imag + other.imag)
```

A third way is to look for what we require from the `other` object, and check that this demand is fulfilled. Mathematically, we require `other` to be a complex or real number, but from a programming point of view, all we demand (in the original `__add__` implementation) is that `other` has `real` and `imag` attributes. To check if an object `a` has an attribute with name stored in the string `attr`, one can use the function

```
hasattr(a, attr)
```

In our context, we need to perform the test

```
if hasattr(other, 'real') and hasattr(other, 'imag'):
```

Our third implementation of the `__add__` method therefore becomes

```
    def __add__(self, other):
        if isinstance(other, (float,int)):
            other = Complex(other)
        elif not (hasattr(other, 'real') and \
                  hasattr(other, 'imag')):
            raise TypeError('other must have real and imag attr.')
        return Complex(self.real + other.real,
                       self.imag + other.imag)
```

The advantage with this third alternative is that we may add instances of class `Complex` and Python's own complex class (`complex`), since all we need is an object with `real` and `imag` attributes.

*Computer Science Discussion.* The presentations of alternative implementations of the `__add__` actually touch some very important computer science topics. In Python, function arguments can refer to objects of any type, and the type of an argument can change during program execution. This feature is known as *dynamic typing* and supported by languages such as Python, Perl, Ruby, and Tcl. Many other languages, C, C++, Java, and C# for instance, restrict a function argument to be of one type, which must be known when we write the program. Any attempt to call the function with an argument of another type is flagged as an error. One says that the language employs *static typing*, since the type cannot change as in languages having dynamic typing. The code snippet

```
a = 6    # a is integer
a = 'b'  # a is string
```

is valid in a language with dynamic typing, but not in a language with static typing.

Our next point is easiest illustrated through an example. Consider the code

```
a = 6
b = '9'
c = a + b
```

The expression `a + b` adds an integer and a string, which is illegal in Python. However, since `b` is the string `'9'`, it is natural to interpret `a + b` as `6 + 9`. That is, if the string `b` is converted to an integer, we may calculate `a + b`. Languages performing this conversion automatically are said to employ *weak typing*, while languages that require the programmer to explicit perform the conversion, as in

```
c = a + float(b)
```

are known to have *strong typing*. Python, Java, C, and C# are examples of languages with strong typing, while Perl and C++ allow weak typing. However, in our third implementation of the `__add__` method, certain types – `int` and `float` – are automatically converted to the right type `Complex`. The programmer has therefore imposed a kind of weak typing in the behavior of the addition operation for complex numbers.

There is also something called *duck typing* where the language only imposes a requirement of some data or methods in the object. The explanation of the term duck typing is the principle: "if it walks like a duck, and quacks like a duck, it's a duck". An operation `a + b` may be valid if `a` and `b` have certain properties that make it possible to add the objects, regardless of the type of `a` or `b`. To enable `a + b` it is in our third implementation of the `__add__` method sufficient that `b` has `real` and `imag` attributes. That is, objects with `real` and `imag` look

like `Complex` objects. Whether they really are of type `Complex` is not considered important in this context.

There is a continuously ongoing debate in computer science which kind of typing that is preferable: dynamic versus static, and weak versus strong. Static and strong typing, as found in Java and C#, support coding safety and reliability at the expense of long and sometimes repetitive code, while dynamic and weak typing support programming flexibility and short code. Many will argue that short code is more reliable than long code, so there is no simple conclusion.

### 7.6.4 Special Methods for "Right" Operands

What happens if we add a `float` and a `Complex` in that order?

```
w = 4.5 + u
```

This statement causes the exception

```
TypeError: unsupported operand type(s) for +: 'float' and 'instance'
```

This time Python cannot find any definition of what the plus operation means with a `float` on the left-hand side and a `Complex` object on the right-hand side of the plus sign. The `float` class was created many years ago without any knowledge of our `Complex` objects, and we are not allowed to extend the `__add__` method in the `float` class to handle `Complex` instances. Nevertheless, Python has a special method `__radd__` for the case where the class instance (`self`) is on the right-hand side of the operator and the `other` object is on the left-hand side. That is, we may implement a possible `float` or `int` plus a `Complex` by

```
    def __radd__(self, other):        # defines other + self
        return self.__add__(other)    # other + self = self + other
```

Similar special methods exist for subtraction, multiplication, and division. For the subtraction operator we need to be a little careful because `other - self`, which is the operation assumed to implemented in `__rsub__`, is not the same as `self.__sub__(other)` (i.e., `self - other`). A possible implementation is

```
    def __sub__(self, other):
        print 'in sub, self=%s, other=%s' % (self, other)
        if isinstance(other, (float,int)):
            other = Complex(other)
        return Complex(self.real - other.real,
                       self.imag - other.imag)

    def __rsub__(self, other):
        print 'in rsub, self=%s, other=%s' % (self, other)
        if isinstance(other, (float,int)):
            other = Complex(other)
        return other.__sub__(self)
```

The `print` statements are inserted to better understand how these methods are visited. A quick test demonstrates what happens:

```
>>> w = u - 4.5
in sub, self=(2, -1), other=4.5
>>> print w
(-2.5, -1)
>>> w = 4.5 - u
in rsub, self=(2, -1), other=4.5
in sub, self=(4.5, 0), other=(2, -1)
>>> print w
(2.5, 1)
```

*Remark.* As you probably realize, there is quite some code to be implemented and lots of considerations to be resolved before we have a class `Complex` for professional use in the real world. Fortunately, Python provides its `complex` class, which offers everything we need for computing with complex numbers. This fact reminds us that it is important to know what others already have implemented, so that we avoid "reinventing the wheel". In a learning process, however, it is a probably a very good idea to look into the details of a class `Complex` as we did above.

### 7.6.5 Inspecting Instances

The purpose of this section is to explain how we can easily look at the contents of a class instance, i.e., the data attributes and the methods. As usual, we look at an example – this time involving a very simple class:

```
class A:
    """A class for demo purposes."""
    def __init__(self, value):
        self.v = value

    def dump(self):
        print self.__dict__
```

The `self.__dict__` attribute is briefly mentioned in Chapter 7.1.5. Every instance is automatically equipped with this attribute, which is a dictionary that stores all the ordinary attributes of the instance (the variable names are keys, and the object references are values). In class `A` there is only one attribute, so the `self.__dict__` dictionary contains one key, `'v'`:

```
>>> a = A([1,2])
>>> a.dump()
{'v': [1, 2]}
```

Another way of inspecting what an instance `a` contains is to call `dir(a)`. This Python function writes out the names of all methods and variables (and more) of an object:

```
>>> dir(a)
'__doc__', '__init__', '__module__', 'dump', 'v']
```

The `__doc__` variable is a docstring, similar to docstrings in functions (Chapter 2.2.7), i.e., a description of the class appearing as a first string right after the `class` headline:

```
>>> a.__doc__
'A class for demo purposes.'
```

The `__module__` variable holds the name of the module in which the class is defined. If the class is defined in the program itself and not in an imported module, `__module__` equals `'__main__'`.

The rest of the entries in the list returned from `dir(a)` correspond to method and attribute names defined by the programmer of the class, in this example the methods `__init__` and `dump`, and the attribute `v`.

Now, let us try to add new variables to an existing instance[8]:

```
>>> a.myvar = 10
>>> a.dump()
{'myvar': 10, 'v': [1, 2]}
>>> dir(a)
['__doc__', '__init__', '__module__', 'dump', 'myvar', 'v']
```

The output of `a.dump()` and `dir(a)` show that we were successful in adding a new variable to this instance on the fly. If we make a new instance, it contains only the variables and methods that we find in the definition of class `A`:

```
>>> b = A(-1)
>>> b.dump()
{'v': -1}
>>> dir(b)
['__doc__', '__init__', '__module__', 'dump', 'v']
```

We may also add new methods to an instance, but this will not be shown here. The primary message of this subsection is two-fold: (i) a class instance is dynamic and allows attributes to be added or removed while the program is running, and (ii) the contents of an instance can be inspected by the `dir` function, and the data attributes are available through the `__dict__` dictionary.

## 7.7 Static Methods and Attributes

Up to now, each instance has its own copy of attributes. Sometimes it can be natural to have attributes that are shared among all instances. For example, we may have an attribute that counts how many instances

[8] This may sound scary and highly illegal to C, C++, Java, and C# programmers, but it is natural and legal in many other languages – and sometimes even useful.

that have been made so far. We can exemplify how to do this in a little class for points $(x, y, z)$ in space:

```
>>> class SpacePoint:
...     counter = 0
...     def __init__(self, x, y, z):
...         self.p = (x, y, z)
...         SpacePoint.counter += 1
```

The `counter` attribute is initialized at the same indentation level as the methods in the class, and the attribute is not prefixed by `self`. Such attributes declared outside methods are shared among all instances and called *static attributes*. To access the `counter` attribute, we must prefix by the classname `SpacePoint` instead of `self`: `SpacePoint.counter`. In the constructor we increase this common counter by 1, i.e., every time a new instance is made the counter is updated to keep track of how many objects we have created so far:

```
>>> p1 = SpacePoint(0,0,0)
>>> SpacePoint.counter
1
>>> for i in range(400):
...     p = SpacePoint(i*0.5, i, i+1)
...
>>> SpacePoint.counter
401
```

The methods we have seen so far must be called through an instance, which is fed in as the `self` variable in the method. We can also make class methods that can be called without having an instance. The method is then similar to a plain Python function, except that it is contained inside a class and the method name must be prefixed by the classname. Such methods are known as *static methods*. Let us illustrate the syntax by making a very simple class with just one static method `write`:

```
>>> class A:
...     @staticmethod
...     def write(message):
...         print message
...
>>> A.write('Hello!')
Hello!
```

As demonstrated, we can call `write` without having any instance of class `A`, we just prefix with the class name. Also note that `write` does not take a `self` argument. Since this argument is missing inside the method, we can never access non-static attributes since these always must be prefixed by an instance (i.e., `self`). However, we can access static attributes, prefixed by the classname.

If desired, we can make an instance and call `write` through that instance too:

```
>>> a = A()
>>> a.write('Hello again')
Hello again
```

Static methods are used when you want a global function, but find it natural to let the function belong to a class and be prefixed with the classname.

## 7.8 Summary

### 7.8.1 Chapter Topics

*Classes.* A class contains attributes (variables) and methods. A first rough overview of a class can be to just list the attributes and methods in a UML diagram as we have done in Figure 7.4 on page 393 for some of the key classes in the present chapter.

Below is a sample class with three attributes (`m`, `M`, and `G`) and three methods (a constructor, `force`, and `visualize`). The class represents the gravity force between two masses. This force is computed by the `force` method, while the `visualize` method plots the force as a function of the distance between the masses.

```
class Gravity:
    """Gravity force between two physical objects."""

    def __init__(self, m, M):
        self.m = m           # mass of object 1
        self.M = M           # mass of object 2
        self.G = 6.67428E-11 # gravity constant, m**3/kg/s**2

    def force(self, r):
        G, m, M = self.G, self.m, self.M
        return G*m*M/r**2

    def visualize(self, r_start, r_stop, n=100):
        from scitools.std import plot, linspace
        r = linspace(r_start, r_stop, n)
        g = self.force(r)
        title='Gravity force: m=%g, M=%g' % (self.m, self.M)
        plot(r, g, title=title)
```

Note that to access attributes inside the `force` method, and to call the `force` method inside the `visualize` method, we must prefix with `self`. Also recall that all methods must take `self`, "this" instance, as first argument, but the argument is left out in calls. The assignment of attributes to a local variable (e.g., `G = self.G`) inside methods is not necessary, but here it makes the mathematical formula easier to read and compare with standard mathematical notation.

This class (found in file `Gravity.py`) can be used to find the gravity force between the moon and the earth:

```
mass_moon = 7.35E+22;  mass_earth = 5.97E+24
gravity = Gravity(mass_moon, mass_earth)
r = 3.85E+8  # earth-moon distance in meters
Fg = gravity.force(r)
print 'force:', Fg
```

*Special Methods.* A collection of special methods, with two leading and trailing underscores in the method names, offers special syntax in Python programs. Table 7.1 on page 392 provides an overview of the most important special methods.

**Table 7.1** Summary of some important special methods in classes. `a` and `b` are instances of the class whose name we set to `A`.

| | |
|---|---|
| `a.__init__(self, args)` | constructor: `a = A(args)` |
| `a.__call__(self, args)` | call as function: `a(args)` |
| `a.__str__(self)` | pretty print: `print a`, `str(a)` |
| `a.__repr__(self)` | representation: `a = eval(repr(a))` |
| `a.__add__(self, b)` | `a + b` |
| `a.__sub__(self, b)` | `a - b` |
| `a.__mul__(self, b)` | `a*b` |
| `a.__div__(self, b)` | `a/b` |
| `a.__radd__(self, b)` | `b + a` |
| `a.__rsub__(self, b)` | `b - a` |
| `a.__rmul__(self, b)` | `b*a` |
| `a.__rdiv__(self, b)` | `b/a` |
| `a.__pow__(self, p)` | `a**p` |
| `a.__lt__(self, b)` | `a < b` |
| `a.__gt__(self, b)` | `a > b` |
| `a.__le__(self, b)` | `a <= b` |
| `a.__ge__(self, b)` | `a => b` |
| `a.__eq__(self, b)` | `a == b` |
| `a.__ne__(self, b)` | `a != b` |
| `a.__bool__(self)` | boolean expression, as in `if a:` |
| `a.__len__(self)` | length of `a` (`int`): `len(a)` |
| `a.__abs__(self)` | `abs(a)` |

### 7.8.2 Summarizing Example: Interval Arithmetics

Input data to mathematical formulas are often subject to uncertainty, usually because physical measurements of many quantities involve measurement errors, or because it is difficult to measure a parameter and one is forced to make a qualified guess of the value instead. In such cases it could be more natural to specify an input parameter by an interval $[a, b]$, which is guaranteed to contain the true value of the parameter. The size of the interval expresses the uncertainty in this parameter. Suppose all input parameters are specified as intervals, what will be the interval, i.e., the uncertainty, of the output data from the formula? This section develops a tool for computing this output uncertainty in

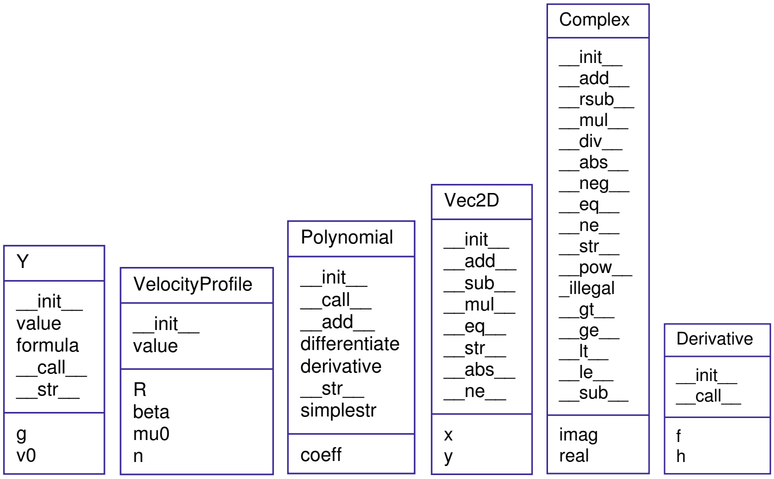



**Fig. 7.4** UML diagrams of some classes described in this chapter.

the cases where the overall computation consists of the standard arithmetic operations.

To be specific, consider measuring the acceleration of gravity by dropping a ball and recording the time it takes to reach the ground. Let the ground correspond to $y = 0$ and let the ball be dropped from $y = y_0$. The position of the ball, $y(t)$, is then[9]

$$y(t) = y_0 - \frac{1}{2}gt^2\,.$$

If $T$ is the time it takes to reach the ground, we have that $y(T) = 0$, which gives the equation $\frac{1}{2}gT^2 = y_0$, with solution

$$g = 2y_0T^{-2}\,.$$

In such experiments we always introduce some measurement error in the start position $y_0$ and in the time taking ($T$). Suppose $y_0$ is known to lie in $[0.99, 1.01]$ m and $T$ in $[0.43, 0.47]$ s, reflecting a 2% measurement error in position and a 10% error from using a stop watch. What is the error in $g$? With the tool to be developed below, we can find that there is a 22% error in $g$.

*Problem.* Assume that two numbers $p$ and $q$ are guaranteed to lie inside intervals,

$$p = [a, b], \quad q = [c, d]\,.$$

The sum $p + q$ is then guaranteed to lie inside an interval $[s, t]$ where $s = a + c$ and $t = b + d$. Below we list the rules of *interval arithmetics*, i.e., the rules for addition, subtraction, multiplication, and division of two intervals:

[9] The formula arises from the solution of Exercise 1.14 when $v_0 = 0$.

1. $p + q = [a + c, b + d]$
2. $p - q = [a - d, b - c]$
3. $pq = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$
4. $p/q = [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$ provided that $[c, d]$ does not contain zero

For doing these calculations in a program, it would be natural to have a new type for quantities specified by intervals. This new type should support the operators `+`, `-`, `*`, and `/` according to the rules above. The task is hence to implement a class for interval arithmetics with special methods for the listed operators. Using the class, we should be able to estimate the uncertainty of two formulas:

1. The acceleration of gravity, $g = 2y_0T^{-2}$, given a 2% uncertainty in $y_0$: $y_0 = [0.99, 1.01]$, and a 10% uncertainty in $T$: $T = [T_m \cdot 0.95, T_m \cdot 1.05]$, with $T_m = 0.45$.
2. The volume of a sphere, $V = \frac{4}{3}\pi R^3$, given a 20% uncertainty in $R$: $R = [R_m \cdot 0.9, R_m \cdot 1.1]$, with $R_m = 6$.

*Solution.* The new type is naturally realized as a class `IntervalMath` whose data consist of the lower and upper bound of the interval. Special methods are used to implement arithmetic operations and printing of the object. Having understood class `Vec2D` from Chapter 7.5, it should be straightforward to understand the class below:

```
class IntervalMath:
    def __init__(self, lower, upper):
        self.lo = float(lower)
        self.up = float(upper)

    def __add__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(a + c, b + d)

    def __sub__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(a - d, b - c)

    def __mul__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(min(a*c, a*d, b*c, b*d),
                            max(a*c, a*d, b*c, b*d))

    def __div__(self, other):
        a, b, c, d = self.lo, self.up, other.lo, other.up
        # [c,d] cannot contain zero:
        if c*d <= 0:
            raise ValueError\
                  ('Interval %s cannot be denominator because '\
                  'it contains zero')
        return IntervalMath(min(a/c, a/d, b/c, b/d),
                            max(a/c, a/d, b/c, b/d))

    def __str__(self):
        return '[%g, %g]' % (self.lo, self.up)
```

The code of this class is found in the file `IntervalMath.py`. A quick demo of the class can go as

```
I = IntervalMath
a = I(-3,-2)
b = I(4,5)
expr = 'a+b', 'a-b', 'a*b', 'a/b'
for e in expr:
    print '%s =' % e, eval(e)
```

The output becomes

```
a+b = [1, 3]
a-b = [-8, -6]
a*b = [-15, -8]
a/b = [-0.75, -0.4]
```

This gives the impression that with very short code we can provide a new type that enables computations with interval arithmetics and thereby with uncertain quantities. However, the class above has severe limitations as shown next.

Consider computing the uncertainty of $aq$ if $a$ is expressed as an interval $[4, 5]$ and $q$ is a number (`float`):

```
a = I(4,5)
q = 2
b = a*q
```

This does not work so well:

```
  File "IntervalMath.py", line 15, in __mul__
    a, b, c, d = self.lo, self.up, other.lo, other.up
AttributeError: 'float' object has no attribute 'lo'
```

The problem is that `a*q` is a multiplication between an `IntervalMath` object `a` and a `float` object `q`. The `__mul__` method in class `IntervalMath` is invoked, but the code there tries to extract the `lo` attribute of `q`, which does not exist since `q` is a `float`.

We can extend the `__mul__` method and the other methods for arithmetic operations to allow for a number as operand – we just convert the number to an interval with the same lower and upper bounds:

```
    def __mul__(self, other):
        if isinstance(other, (int, float)):
            other = IntervalMath(other, other)
        a, b, c, d = self.lo, self.up, other.lo, other.up
        return IntervalMath(min(a*c, a*d, b*c, b*d),
                            max(a*c, a*d, b*c, b*d))
```

Looking at the formula $g = 2y_0T^{-2}$, we run into a related problem: now we want to multiply 2 (`int`) with $y_0$, and if $y_0$ is an interval, this multiplication is not defined among `int` objects. To handle this case, we need to implement an `__rmul__(self, other)` method for doing `other*self`, as explained in Chapter 7.6.4:

```
    def __rmul__(self, other):
        if isinstance(other, (int, float)):
            other = IntervalMath(other, other)
        return other*self
```

Similar methods for addition, subtraction, and division must also be included in the class.

Returning to $g = 2y_0T^{-2}$, we also have a problem with $T^{-2}$ when $T$ is an interval. The expression `T**(-2)` invokes the power operator (at least if we do not rewrite the expression as `1/(T*T)`), which requires a `__pow__` method in class `IntervalMath`. We limit the possibility to have integer powers, since this is easy to compute by repeated multiplications:

```
    def __pow__(self, exponent):
        if isinstance(exponent, int):
            p = 1
            if exponent > 0:
                for i in range(exponent):
                    p = p*self
            elif exponent < 0:
                for i in range(-exponent):
                    p = p*self
                p = 1/p
            else:   # exponent == 0
                p = IntervalMath(1, 1)
            return p
        else:
            raise TypeError('exponent must int')
```

Another natural extension of the class is the possibility to convert an interval to a number by choosing the midpoint of the interval:

```
>>> a = IntervalMath(5,7)
>>> float(a)
6
```

`float(a)` calls `a.__float__()`, which we implement as

```
    def __float__(self):
        return 0.5*(self.lo + self.up)
```

A `__repr__` method returning the right syntax for recreating the present instance is also natural to include in any class:

```
    def __repr__(self):
        return '%s(%g, %g)' % \
               (self.__class__.__name__, self.lo, self.up)
```

We are now in a position to test out the extended class `IntervalMath`.

```
>>> g = 9.81
>>> y_0 = I(0.99, 1.01)      # 2% uncertainty
>>> Tm = 0.45                # mean T
>>> T = I(Tm*0.95, Tm*1.05)  # 10% uncertainty
>>> print T
```

```
[0.4275, 0.4725]
>>> g = 2*y_0*T**(-2)
>>> g
IntervalMath(8.86873, 11.053)
>>> # computing with mean values:
>>> T = float(T)
>>> y = 1
>>> g = 2*y_0*T**(-2)
>>> print '%.2f' % g
9.88
```

Another formula, the volume $V = \frac{4}{3}\pi R^3$ of a sphere, shows great sensitivity to uncertainties in $R$:

```
>>> Rm = 6
>>> R = I(Rm*0.9, Rm*1.1)   # 20 % error
>>> V = (4./3)*pi*R**3
>>> V
IntervalMath(659.584, 1204.26)
>>> print V
[659.584, 1204.26]
>>> print float(V)
931.922044761
>>> # compute with mean values:
>>> R = float(R)
>>> V = (4./3)*pi*R**3
>>> print V
904.778684234
```

Here, a 20% uncertainty in $R$ gives almost 60% uncertainty in $V$, and the mean of the $V$ interval is significantly different from computing the volume with the mean of $R$.

The complete code of class `IntervalMath` is found in `IntervalMath.py`. Compared to the implementations shown above, the real implementation in the file employs some ingenious constructions and help methods to save typing and repeating code in the special methods for arithmetic operations.

## 7.9 Exercises

**Exercise 7.1.** *Make a function class.*

Make a class `F` that implements the function

$$f(x; a, w) = e^{-ax}\sin(wx)\,.$$

A `value(x)` method computes values of $f$, while $a$ and $w$ are class attributes. Test the class with the following main program:

```
from math import *
f = F(a=1.0, w=0.1)
print f.value(x=pi)
f.a = 2
print f.value(pi)
```

Name of program file: `F.py`. ◇

**Exercise 7.2.** *Make a very simple class.*

Make a class `Simple` with one attribute `i`, one method `double`, which replaces the value of `i` by `i+i`, and a constructor that initializes the attribute. Try out the following code for testing the class:

```
s1 = Simple(4)
for i in range(4):
    s1.double()
print s1.i

s2 = Simple('Hello')
s2.double();  s2.double()
print s2.i
s2.i = 100
print s2.i
```

Before you run this code, convince yourself what the output of the `print` statements will be. Name of program file: `Simple.py`. ◇

**Exercise 7.3.** *Extend the class from Ch. 7.2.1.*

Add an attribute `transactions` to the `Account` class from Chapter 7.2.1. The new attribute counts the number of transactions done in the `deposit` and `withdraw` methods. The total number of transactions should be printed in the `dump` method. Write a simple test program to demonstrate that `transaction` gets the right value after some calls to `deposit` and `withdraw`. Name of program file: `Account2.py`. ◇

**Exercise 7.4.** *Make classes for a rectangle and a triangle.*

The purpose of this exercise is to create classes like class `Circle` from Chapter 7.2.3 for representing other geometric figures: a rectangle with width $W$, height $H$, and lower left corner $(x_0, y_0)$; and a general triangle specified by its three vertices $(x_0, y_0)$, $(x_1, y_1)$, and $(x_2, y_2)$ as explained in Exercise 2.17. Provide three methods: `__init__` (to initialize the geometric data), `area`, and `circumference`. Name of program file: `geometric_shapes.py`. ◇

**Exercise 7.5.** *Make a class for straight lines.*

Make a class `Line` whose constructor takes two points `p1` and `p2` (2-tuples or 2-lists) as input. The line goes through these two points (see function `line` in Chapter 2.2.7 for the relevant formula of the line). A `value(x)` method computes a value on the line at the point `x`. Here is a demo in an interactive session:

```
>>> from Line import Line
>>> line = Line((0,-1), (2,4))
>>> print line.value(0.5), line.value(0), line.value(1)
0.25 -1.0 1.5
```

Name of program file: `Line.py`. ◇

**Exercise 7.6.** *Improve the constructor in Exer. 7.5.*

The constructor in class `Line` in Exercise 7.5 takes two points as arguments. Now we want to have more flexibility in the way we specify

a straight line: we can give two points, a point and a slope, or a slope and the line's interception with the $y$ axis. Hint: Let the constructor take two arguments `p1` and `p2` as before, and test with `isinstance` whether the arguments are `float` or `tuple/list` to determine what kind of data the user supplies:

```
if isinstance(p1, (tuple,list)) and isinstance(p2, (float,int)):
    self.a = p2                   # p2 is slope
    self.b = p1[1] - p2*p1[0]     # p1 is a point
elif ...
```

Name of program file: `Line2.py`. ◇

**Exercise 7.7.** *Make a class for quadratic functions.*

Consider a quadratic function $f(x; a, b, c) = ax^2 + bx + c$. Make a class `Quadratic` for representing $f$, where $a$, $b$, and $c$ are attributes, and the methods are

1. `value` for computing a value of $f$ at a point $x$,
2. `table` for writing out a table of $x$ and $f$ values for $n$ $x$ values in the interval $[L, R]$,
3. `roots` for computing the two roots.

Name of program file: `Quadratic.py`. ◇

**Exercise 7.8.** *Make a class for linear springs.*

To elongate a spring a distance $x$, one needs to pull the spring with a force $kx$. The parameter $k$ is known as the spring constant. The corresponding potential energy in the spring is $\frac{1}{2}kx^2$.

Make a class for springs. Let the constructor store $k$ as a class attribute, and implement the methods `force(x)` and `energy(x)` for evaluating the force and the potential energy, respectively.

The following function prints a table of function values for an arbitrary mathematical function `f(x)`. Demonstrate that you can send the `force` and `energy` methods as the `f` argument to `table`.

```
def table(f, a, b, n, heading=''):
    """Write out f(x) for x in [a,b] with steps h=(b-a)/n."""
    print heading
    h = (b-a)/float(n)
    for i in range(n+1):
        x = a + i*h
        print 'function value = %10.4f at x = %g' % (f(x), x)
```

Name of program file: `Spring.py`. ◇

**Exercise 7.9.** *Implement Lagrange's interpolation formula.*

Given $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, the following polynomial of degree $n$ goes through all the points:

$$p_L(x) = \sum_{k=0}^{n} y_k L_k(x), \tag{7.14}$$

$$L_k(x) = \frac{\prod_{i=0,i\neq k}^{n}(x - x_i)}{\prod_{j=0,j\neq k}^{n}(x_k - x_j)}\,.$$

Make a class `Lagrange` whose constructor takes the $n+1$ points as input in the form of a list of points. A special method `__call__` evaluates the polynomial $p_L(x)$ at a point $x$.

To verify the program, we observe that $L_k(x_k) = 1$ and that $L_k(x_i) = 0$ for $i \neq k$ such that $p_L(x_k)$ equals $y_k$. Write a method `_verify` in class `Lagrange` that computes $|p_L(x_k) - y_k|$ at all points and checks that the value is approximately zero (e.g., less than $10^{-14}$). Generate 5 points along the curve $y = \sin(x)$ for $x \in [0, \pi]$ and call `_verify`.

The formula (7.14) is called Lagrange's interpolation formula, because given $n+1$ points, the formula makes it possible to evaluate function values between the points[10], and at the points we recover the $y_k$ values. Sometimes the polynomial $p_L$ exhibit a strange behavior, which we now want to explore. Generate $n+1$ points for $n = 3, 5, 7, 11$ along the curve $y = |x|$, with $x \in [-2, 2]$, and plot $p_L(x)$ for each $n$ value (use many more than $n+1$ points to plot the polynomials - otherwise you just get straight lines between the $n+1$ points). Also plot the $n+1$ points as circles. Observe from the plot that $p_L$ always goes through the points, but as the number of points increases, $p_L$ oscillates more and more, and the approximation of $p_L$ to the curve $|x|$ is getting poorer. Make an additional plot of $p_L$ corresponding to $n = 13$ and $n = 20$ and observe how the amplitude of the oscillations increase with increasing $n$. These oscillations are undesired, and much research has historically been focused on methods that do not result in "strange oscillations" when fitting a polynomial to a set of points.

Name of program file: `Lagrange_polynomial.py`. ◇

**Exercise 7.10.** *A very simple "Hello, World!" class.*

Make a class that can only do one thing: `print a` writes "Hello, World!" to the screen, when `a` is an instance of the class. Name of program file: `HelloWorld.py`. ◇

**Exercise 7.11.** *Use special methods in Exer. 7.1.*

Modify the class from Exercise 7.1 such that the following code works:

```
f = F2(1.0, 0.1)
print f(pi)
f.a = 2
print f(pi)
print f
```

Name of program file: `F2.py`. ◇

[10] This is called interpolation.

**Exercise 7.12.** *Modify a class for numerical differentiation.*

Make the two attributes `h` and `f` of class `Derivative` from Chapter 7.3.2 protected as explained in Chapter 7.2.1. That is, prefix `h` and `f` with an underscore to tell users that these attributes should be accessed directly. Add two methods `get_precision()` and `set_precision(h)` for reading and changing the $h$ parameter in the numerical differentiation formula. Apply the modified class to make a table of the approximation error of the derivative of $f(x) = \ln x$ for $x = 1$ and $h = 2^{-k}$, $k = 1, 5, 9, 13 \ldots, 45$. Name of program file: `geometric_shapes.py`. ◇

**Exercise 7.13.** *Make a class for nonlinear springs.*

This exercise is a generalization of Exercise 7.8. To elongate a spring a distance $x$, one needs a force $f(x)$. If $f$ is linear, as in Exercise 7.8 ($f(x) = kx$), we speak of a linear spring, otherwise the spring is nonlinear. The potential energy stored in the elongated spring, arising from the work done by the force, is given by the integral $\int_0^x f(t)dt$.

Make a class for nonlinear springs which has the methods `force(x)` and `energy(x)` for computing the force and potential energy respectively. Make use of class `Integral` from Chapter 7.3.3 in the `energy` method to compute the integral numerically.

Demonstrate the class for a linear spring $f(x) = kx$. Also demonstrate the class for a nonlinear spring $f(x) = a\sin(x)$. Implement $f$ in both cases as a class with $k$ or $a$ as attribute. Name of program file: `Spring_nonlinear.py`. ◇

**Exercise 7.14.** *Extend the class from Ch. 7.2.1.*

As alternatives to the `deposit` and `withdraw` methods in class `Account` class from Chapter 7.2.1, we could use `+=` for `deposit` and `-=` for `withdraw`. The special methods `__iadd__` and `__isub__` implement the `+=` and `-=` operators, respectively. For instance, `a -= p` implies a call to `a.__isub__(p)`. One important feature of `__iadd__` and `__isub__` is that they must return `self` to work properly, cf. the documentation of these methods in the Python Language Reference (not to be confused with the Python Library Reference).

Implement the `+=` and `-=` operators, a `__str__` method, and preferably a `__repr__` method. Provide, as always, some code to test that the new methods work as intended. Name of program file: `Account3.py`. ◇

**Exercise 7.15.** *Implement a class for numerical differentation.*

A widely used formula for numerical differentiation of a function $f(x)$ takes the form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \,. \tag{7.15}$$

This formula usually gives more accurate derivatives than (7.1) because it applies a centered, rather than a one-sided, difference.

The goal of this exercise is to use the formula (7.15) to automatically differentiate a mathematical function $f(x)$ implemented as a Python function `f(x)`. More precisely, the following code should work:

```
def f(x):
    return 0.25*x**4

df = Central(f)  # make function-like object df
# df(x) computes the derivative of f(x) approximately:
for x in (1, 5, 10):
    df_value = df(x)  # approx value of derivative of f at point x
    exact = x**3      # exact value of derivative
    print "f'(%d)=%g  (error=%.2E)" % (x, df_value, exact-df_value)
```

Implement class `Central` and test that the code above works. Include an optional argument `h` to the constructor in class `Central` so that one can specify the value of $h$ in the approximation (7.15). Apply class `Central` to produce a table of the derivatives and the associated approximation errors for $f(x) = \ln x$, $x = 10$, and $h = 0.5, 0.1, 10^{-3}, 10^{-5}, 10^{-7}, 10^{-9}, 10^{-11}$. Collect class `Central` and the two applications of the class in the same file, but organize the file as a module so that class `Central` can be imported in other files. Name of program file: `Central.py`. ⋄

**Exercise 7.16.** *Verify a program.*

Consider this program file for computing a backward difference approximation to the derivative of a function `f(x)`:

```
from math import *

class Backward:
    def __init__(self, f, h=e-9):
        self.f, self.h = f, h
    def __call__(self, x):
        h, f = self.h, self.f
        return (f(x) - f(x-h))/h  # finite difference

dsin = Backward(sin)
e = dsin(0) - cos(0); print 'error:', e
dexp = Backward(exp, h=e-7)
e = dexp(0) - exp(0); print 'error:', e
```

The output becomes

```
error: -1.00023355634
error: 371.570909212
```

Is the approximation that bad, or are there bugs in the program? ⋄

**Exercise 7.17.** *Test methods for numerical differentation.*

Make a function `table(f, x, hlist, dfdx=None)` for writing out a nicely formatted table of the errors in the numerical derivative of a function `f(x)` at point `x` using the two formulas (7.1) and 7.15 and their implementations in classes `Derivative` (from Chapter 7.3.2), and `Central` (from Exercise 7.15). The first column in the table shows a

list of $h$ values (`hlist`), while the two next columns contain the corresponding errors arising from the two numerical approximations of the first derivative. The `dfdx` argument may hold a Python function that returns the exact derivative. Write out an additional column with the exact derivative if `dfdx` is given (i.e., not `None`).

Call `table` for each of the functions $x^2$, $\sin^6(\pi x)$, and $\tanh(10x)$, and the $x$ values 0 and 0.25. Can you see from the errors in the tables which of the three approximations that seems to have the overall best performance in these examples? Plot the three functions on $[-1, 1]$ and try to understand the behavior of the various approximations from the plots. Name of program file: `Derivative_comparisons.py`. ◇

**Exercise 7.18.** *Make a class for summation of series.*

Our task in this exercise is to calculate a sum $S(x) = \sum_{k=M}^{N} f_k(x)$, where $f_k(x)$ is a term in a sequence which is assumed to decrease in absolute value. In class `Sum`, for computing $S(x)$, the constructor requires the following three arguments: $f_k(x)$ as a function `f(k, x)`, $M$ as an `int` object `M`, and $N$ as an `int` object `N`. A `__call__` method computes and returns $S(x)$. The next term in the series, $f_{N+1}(x)$, should be computed and stored as an attribute `first_neglected_term`. Here is an example where we compute $S(x) = \sum_{k=0}^{N} (-x)^k$:

```
def term(k, x):  return (-x)**k

S = Sum(term, M=0, N=100)
x = 0.5
print S(x)
# print the value of the first neglected term from last S(x) comp.:
print S.first_neglected_term
```

Calculate by hand what the output of this test becomes, and use it to verify your implementation of class `Sum`.

Apply class `Sum` to compute the Taylor polynomial approximation for $\sin x$ at $x = \pi, 30\pi$ and $N = 5, 10, 20$. Compute the error and compare with the first neglected term $f_{N+1}(x)$. Present the result in nicely formatted tables. Repeat such calculations for the Taylor polynomial for $e^{-x}$ at $x = 1, 3, 5$ and $N = 5, 10, 20$. Also demonstrate how class `Sum` can be used to calculate the sum (2.1) on page 78 (choose $x = 2, 5, 10$ and $N = 5, 10, 20$). Formulas for the Taylor polynomials can be looked up in Exercise 4.18. Name of program file: `Sum.py`. ◇

**Exercise 7.19.** *Apply the differentiation class from Ch. 7.3.2.*

Use class `Derivative` from page 358 to calculate the derivative of the function $v$ on page 230 with respect to the parameter $n$, i.e., $\frac{dv}{dn}$. Choose $\beta/\mu_0 = 50$ and $r/R = 0.5$, and compare the result with the exact derivative. Hint: Make a class similar to `VelocityProfile` on page 346, but provide `r` as a parameter to the constructor, instead of `n`, and let `__call__` take `n` as parameter. Exercise 9.6 suggests a more advanced

technique that can also be applied here to implement $v$ as a function of $n$. Name of program file: `VelocityProfile_deriv.py`. ◇

**Exercise 7.20.** *Use classes for computing inverse functions.*

Chapter 5.1.10 describes a method and implementation for computing the inverse function of a given function. The purpose of the present exercise is to improve the implementation in Chapter 5.1.10 by introducing classes. This allows a program that is more flexible with respect to the way we can specify the function to be inverted.

Implement the `F` and `dFdx` functions from Chapter 5.1.10 as classes to avoid relying on global variables for `h`, `xi`, etc. Also introduce a class `InverseFunction` to run the complete algorithm and store the `g` array (from Chapter 5.1.10) as an array attribute `values`. Here is a typical use of class `InverseFunction`:

```
>>> from InverseFunction import InverseFunction as I
>>> from scitools.std import *
>>> def f(x):
...     return log(x)
...
>>> x = linspace(1, 5, 101)
>>> f_inv = I(f, x)
>>> plot(x, f(x), x, f_inv.values)
```

Check, in the constructor, that `f` is monotonically increasing or decreasing over the set of coordinates (`x`). Errors may occur in the computations because Newton's method might divide by zero or diverge. Make sure sensible error messages are reported in those cases.

A `__call__` in class `InverseFunction` should should evaluate the inverse function at an arbitrary point `x`. This is somewhat challenging, however, since we only have the inverse function at discrete points along its curve. With aid of a function `wrap2callable` from `scitools.std` one can turn $(x, y)$ points on a curve, stored in arrays `x` and `y`, into a (piecewise) *continuous* Python function `q(x)` by:

```
q = wrap2callable((x, y))
```

In a sense, the `wrap2callable` draws lines between the discrete points to form the resulting continuous function. Use `wrap2callable` to make `__call__` evaluate the inverse function at any point in the interval from `x[0]` to `x[-1]`. Name of program file: `InverseFunction.py`. ◇

**Exercise 7.21.** *Vectorize a class for numerical integration.*

Use a vectorized implementation of the Trapezoidal rule in class `Integral` from Chapter 7.3.3. Name of program file: `Integral_vec.py`. ◇

**Exercise 7.22.** *Speed up repeated integral calculations.*

The observant reader may have noticed that our `Integral` class from Chapter 7.3.3 is very inefficient if we want to tabulate or plot a function

$F(x) = \int_a^x f(x)$ for several consecutive values of $x$, say $x_0 < x_1 < \cdots < x_n$. Requesting $F(x_k)$ will recompute the integral computed as part of $F(x_{k-1})$, and this is of course waste of computer work.

Modify the `__call__` method such that if `x` is an array, assumed to contain coordinates of increasing value: $x_0 < x_1 < \cdots < x_n$, the method returns an array with $F(x_0), F(x_1), \ldots, F(x_n)$. Make sure you are not doing any redundant calculations. (Hint: The $F$ values can be efficiently calculated by one pass through the `x` array if you store intermediate integral approximations in the Trapezoidal algorithm.) Name of program file: `Integral_eff.py`. ◇

**Exercise 7.23.** *Solve a simple ODE in two ways.*

The purpose of this exercise is to solve the ODE problem $u' = u/10$, $u(0) = 0.2$, for $t \in [0, 20]$. Use both the `ForwardEuler` *function* and the `ForwardEuler` class from Chapter 7.4, both with $\Delta t = 1$. Check that the results produced by the two equivalent methods coincide (you may use the `float_eq` function, see Exercise 2.51). Plot the solution. Name of program file: `simple_ODE.py`. ◇

**Exercise 7.24.** *Solve the ODE (B.36).*

Use the `ForwardEuler` class from Chapter 7.4 to solve the ODE (B.36) described in Exercise 9.25 on page 554. Use the parameters as described in Exercise 9.25 and simulate for 200 seconds. Plot the solution. Name of program file: `tank_ODE_FEclass.py`. ◇

**Exercise 7.25.** *Simulate a falling or rising body in a fluid.*

A body moving vertically through a fluid (liquid or gas) is subject to three different types of forces:

1. the gravity force $F_g = -mg$, where $m$ is the mass of the body and $g$ is the acceleration of gravity;
2. the drag force[11] $F_d = -\frac{1}{2}C_D\varrho A|v|v$ (see also Exercise 1.10), where $C_D$ is a dimensionless drag coefficient depending on the body's shape, $\varrho$ is the density of the fluid, $A$ is the cross-sectional area (produced by a cutting plane $y =$ const through the thickest part of the body), and $v$ is the velocity;
3. the uplift or buoyancy force ("Archimedes force") $F_b = \varrho gV$, where $V$ is the volume of the body.

Newton's second law applied to the body says that the sum of these forces must equal the mass of the body times its acceleration $a$:

$$F_g + F_d + F_b = ma,$$

which gives

[11] Roughly speaking, the $F_d$ formula is suitable for medium to high velocities, while for very small velocities, or very small bodies, $F_d$ is proportional to the velocity, not the velocity squared, see [11].

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho gV = ma\,.$$

The unknowns here are $v$ and $a$, i.e., we have two unknowns but only one equation. From kinetmatics in physics we know that the acceleration is the time derivative of the velocity: $a = dv/dt$. This is our second equation. We can easily eliminate $a$ and get a single differential equation for $v$:

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho gV = m\frac{dv}{dt}\,.$$

A small rewrite of this equation is handy: We express $m$ as $\varrho_b V$, where $\varrho_b$ is the density of the body, and we isolate $dv/dt$ on the left-hand side,

$$\frac{dv}{dt} = -g\left(1 - \frac{\varrho}{\varrho_b}\right) - \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,. \tag{7.16}$$

This differential equation must be accompanied by an initial condition: $v(0) = v_0$. Make a program for solving (7.16) numerically, utilizing tools from Chapter 7.4. Implement the right-hand side of (7.16) in the `__call__` method of a class where the parameters $g$, $\varrho$, $\varrho_b$, $C_D$, $A$, and $V$ are attributes.

To verify the program, assume a heavy body in air such that the $F_b$ force can be neglected, and assume a small velocity such that the air resistance $F_d$ can also be neglected. Setting $\varrho = 0$ removes both these terms from the equation. The motion is then described by (1.1), and the exact velocity is $v(t) = y'(t) = v_0 - gt$. See how well the program reproduces this simple solution.

After the program is verified, we are ready to run two real examples and plot the evolution of $v$:

1. *Parachute jumper.* The point is to compute the motion of a parachute jumper in free fall before the parachute opens. We set the density of the human body as $\varrho_b = 1003$ kg/m$^3$ and the mass as $m = 80$ kg, implying $V = m/\varrho_b = 0.08$ m$^3$. We can base the cross-sectional area $A$ on the height 1.8 m and a width of 50 cm, giving giving $A \approx \pi R^2 = 0.9$ m$^2$. The density of air decreases with height, and we here use the value 0.79 kg/m$^3$ which is relevant for about 5000 m height. The $C_D$ coefficient can be set as 0.6. Start with $v_0 = 0$.
2. *Rising ball in water.* A ball with the size of a soccer ball is placed in deep water, and we seek to model its motion upwards. Contrary to the former example, where the buoyancy force $F_b$ is very small, $F_b$ is now the driving force, and the gravity force $F_g$ is small. Set $A = \pi a^2$ with $a = 11$ cm, the mass of the ball is 0.43 kg, the density of water is 1000 kg/m$^3$, and $C_D$ is 0.2. Start with $v_0 = 0$ and see how the ball rises.

Name of program file: `body_in_fluid.py`. ◇

**Exercise 7.26.** *Check the solution's limit in Exer. 7.25.*

The solution of (7.16) often tends to a constant velocity, called the terminal velocity. This happens when the sum of the forces, i.e., the right-hand side in (7.16) vanishes. Compute the formula for the terminal velocity by hand. Modify class `ForwardEuler` from Chapter 7.4 so that the time loop is run as long as $k \leq N$ *and* a user-defined function `terminate(u, t, k)` is `False`, where `u` is the list of $u_i$ values for $i = 0, 1, \ldots, k$, `t` is the corresponding list of time points, and `k` is the current time step counter. The `terminate` function can be given as an extra argument to `solve`. Here is an outline of the modified class `ForwardEuler`:

```
class ForwardEuler:
    ...
    def solve(self, N, terminate=None):
        if terminate is None:
            terminate = lambda u, t, k: False
        self.k = 0
        while self.k < N and \
            not terminate(self.u, self.t, self.k):
            unew = self.advance()
            ...
            self.k += 1
```

In your implementation of `terminate`, stop the simulation when a constant velocity is reached, that is, when $|v(t_n) - v(t_{n-1})| \leq \epsilon$, where $\epsilon$ is a small number. Run a series of $\Delta t$ values and make a graph of the terminal velocity as a function of $\Delta t$ for the two cases in Exercise 7.25. Plot a horizontal line with the exact terminal velocity. Name of program file: `body_in_fluid_termvel.py`. ◇

**Exercise 7.27.** *Implement the modified Euler method; function.*

The following numerical method for solving $u' = f(u, t)$, $u(0) = u_0$, is known as the modified Euler method:

$$v_q = u_k + \frac{1}{2}\Delta t\left(f(v_{q-1}, t_{k+1}) + f(u_k, t_k)\right),$$

$$q = 1, \ldots, N,\ v_0 = u_k \qquad (7.17)$$

$$u_{k+1} = v_N\,. \qquad (7.18)$$

At each time level, we run the formula (7.17) $N$ times, and the $n$-th value $v_N$ is the new value of $u$, i.e., $u_{k+1}$. Setting $N = 1$ recovers the Forward Euler scheme, while $N$ corresponds to a 2nd-order Runge-Kutta scheme. We can either fix the value of $N$, or we can repeat (7.18) until the change in $v_q$ is small, that is, until $|v_q - v_{q-1}| < \epsilon$, where $\epsilon$ is a small value. Fixing $N$ is sufficient in this exercise.

Implement (7.17)–(7.18) as a function

```
EulerMidpoint_method(f, dt, u0, T, N)
```

where `f` is a Python implementation of $f(u, t)$, `dt` is the time step $\Delta t$, `u0` is the initial condition $u(0)$, `T` is the final time of the simulation, and `N` is the parameter $N$ in the method (7.17). The `EulerMidpoint_method` should return two arrays: $u_0, \ldots, u_N$ and $t_0, \ldots, t_N$. To verify the implementation, calculate by hand $u_1$ and $u_2$ when $N = 2$ for the ODE $u' = -2u$, $u(0) = 1$, with $\Delta t = 1/2$. Compare your hand calculations with the results of the program. Thereafter, run the program for the same ODE problem but with $\Delta t = 0.1$ and $T = 2$. Name of program file: `ModifiedEuler_func.py`. ◇

**Exercise 7.28.** *Implement the modified Euler method; class.*

The purpose of this exercise is to implement the modified Euler method, defined in Exercise 7.28, in a class like the `ForwardEuler` class from Chapter 7.4. Create a module containing the class and a test function demonstrating the use:

```
def _test():
    def f(u, t):
        return -2*u

    dt = 0.1
    N = 4
    method = ModifiedEuler(f, dt, N)
    method.set_initial_condition(1)
    T = 1.5
    u, t = method.solve(T)
    from scitools.std import plot
    plot(t, u)
```

Call the `_test` function from the test block in the module file. Name of program file: `ModifiedEuler1.py`. ◇

**Exercise 7.29.** *Increase the flexibility in Exer. 7.28.*

The constructor of class `ModifiedEuler` in Exercise 7.28 takes an `N` parameter with a fixed value. We can, as explained in Exercise 7.27, either fix $N$ or introduce an $\epsilon$ and iterate until the change in $|v_q - v_{q-1}|$ is less than $\epsilon$. Modify the constructor so that we can give both $N$ and $\epsilon$. We compute a new $v_q$ as long as $q \leq N$ or $|v_q - v_{q-1}| > \epsilon$. Let $N = 20$ and $\epsilon = 10^{-6}$ by default. Name of program file: `ModifiedEuler2.py`. ◇

**Exercise 7.30.** *Solve an ODE specified on the command line.*

To solve an ODE, we want to make a program `odesolver.py` which accepts an PDE problem to be specified on the command line. The typical syntax of running the programs looks like

```
odesolver.py f u0 dt T
```

where `f` is the right-hand side $f(u, t)$ specified as a string formula (to be converted to a `StringFunction` object), `u0` is the initial condition, `dt` is the time step, and `T` is the final time of the simulation. A curve plot of the solution versus time should be produced and stored in a file `plot.png`. Make the `odesolver.py` program, using any numerical

method that you like. Hint: You may start with the programs in Chapter 7.4 and add a command-line user interface. Name of program file: `odesolver.py`. ◇

**Exercise 7.31.** *Apply a polynomial class.*

The Taylor polynomial of degree $N$ for the exponential function $e^x$ is given by

$$p(x) = \sum_{k=0}^{N} \frac{x^k}{k!}\,.$$

Use class Polynomial from page 365 to represent this $p(x)$ polynomial for a given $N$. Choose $x = 2$ and $N = 5, 25, 50, 100$, and compare the accuracy of the result with `math.exp(2)`. Name of program file: `Polynomial_exp.py`. ◇

**Exercise 7.32.** *Find a bug in a class for polynomials.*

Go through this alternative implementation of class `Polynomial` from page 365 and explain each line in detail:

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        return sum([c*x**i for i, c in enumerate(self.coeff)])

    def __add__(self, other):
        maxlength = max(len(self), len(other))
        # extend both lists with zeros to this maxlength:
        self.coeff += [0]*(maxlength - len(self.coeff))
        other.coeff += [0]*(maxlength - len(other.coeff))
        sum_coeff = self.coeff
        for i in range(maxlength):
            sum_coeff[i] += other.coeff[i]
        return Polynomial(sum_coeff)
```

Look up what the `enumerate` function does (use the Python Library Reference). Write this code in a file, and demonstrate that adding two polynomials does not work. Find the bug and correct it. Name of program file: `Polynomial_error.py`. ◇

**Exercise 7.33.** *Subtraction of polynomials.*

Implement the special method `__sub__` in class `Polynomial` from page 365. Name of program file: `Polynomial_sub.py`. ◇

**Exercise 7.34.** *Represent a polynomial by an array.*

Introduce a Numerical Python array for `self.coeff` in class `Polynomial` from page 365. Go through the class code and run the statements in the `_test` method in the `Polynomial.py` file to locate which statements that need to be modified because `self.coeff` is an array and not a list. Name of program file: `Polynomial_array1.py`. ◇

**Exercise 7.35.** *Vectorize a class for polynomials.*

Introducing an array instead of a list in class `Polynomial`, as suggested in Exercise 7.34, does not enhance the implementation. A real enhancement arise when the code is vectorized, i.e., when loops are replaced by operations on whole arrays.

First, vectorize the `__add__` method by adding the common parts of the coefficients arrays and then appending the rest of the longest array to the result (appending an array `a` to an array `b` is done by `concatenate(a, b)`).

Second, vectorize the `__call__` method by observing that evaluation of a polynomial, $\sum_{i=0}^{n-1} c_i x^i$, can be computed as the inner product of two arrays: $(c_0, \ldots, c_{n-1})$ and $(x^0, x^1, \ldots, x^{n-1})$. The latter array can be computed by `x**p`, where `p` is an array with powers $0, 1, \ldots, n-1$ made by `linspace` or `arange`.

Third, the `differentiate` method can be vectorized by the statements

```
n = len(self.coeff)
self.coeff[:-1] = linspace(1, n-1, n-1)*self.coeff[1:]
self.coeff = self.coeff[:-1]
```

Show by hand calculations in a case where `n` is 3 that the vectorized statements produce the same result as the original `differentiate` method.

The `__mul__` method is more challenging to vectorize so you may leave this unaltered. Check that the vectorized versions of `__add__`, `__call__`, and `differentiate` work by comparing with the scalar code from Exercise 7.34 or the original, list-based `Polynomial` class. Name of program file: `Polynomial_array2.py`. ◇

**Exercise 7.36.** *Use a dict to hold polynomial coefficients; add.*

Use a dictionary for `self.coeff` in class `Polynomial` from page 365. The advantage with a dictionary is that only the nonzero coefficients need to be stored. Let `self.coeff[k]` hold the coefficient of the $x^k$ term. Implement a constructor and the `__add__` method. Exemplify the implementation by adding $x - 3x^{100}$ and $x^{20} - x + 4x^{100}$. Name of program file: `Polynomial_dict1.py`. ◇

**Exercise 7.37.** *Use a dict to hold polynomial coefficients; mul.*

Extend the class in Exercise 7.36 with a `__mul__` method. First, study the algorithm in Chapter 7.3.7 for the `__mul__` method when the coefficients are stored in lists. Then modify the algorithm to work with dictionaries. Implement the algorithm and exemplify it by multiplying $x - 3x^{100}$ and $x^{20} - x + 4x^{100}$. Name of program file: `Polynomial_dict2.py`. ◇

**Exercise 7.38.** *Extend class Vec2D to work with lists/tuples.*

The `Vec2D` class from Chapter 7.5 supports addition and subtraction, but only addition and subtraction of two `Vec2D` objects. Sometimes we

would like to add or subtract a point that is represented by a list or a tuple:

```
u = Vec2D(-2, 4)
v = u + (1,1.5)
w = [-3, 2] - v
```

That is, a list or a tuple must be allowed in the right or left operand. Use ideas from Chapters 7.6.3 and 7.6.4 to implement this extension. Name of program file: `Vec2D_lists.py`. ◇

**Exercise 7.39.** *Use NumPy arrays in class Vec2D.*

The internal code in class `Vec2D` from Chapter 7.5 can be valid for vectors in any space dimension if we represent the vector as a NumPy array in the class instead of seperate variables `x` and `y` for the vector components. Make a new class `Vec` where you apply NumPy functionality in the methods. The constructor should be able to treat all the following ways of initializing a vector:

```
a = array([1, -1, 4], float)  # numpy array
v = Vec(a)
v = Vec([1, -1, 4])           # list
v = Vec((1, -1, 4))           # tuple
v = Vec(1, -1)                # coordinates
```

We will provide some helpful advice. In the constructor, use variable number of arguments as described in Appendix E.5. All arguments are then available as a tuple, and if there is only one element in the tuple, it should be an array, list, or tuple you can send through `asarray` to get a NumPy array. If there are many arguments, these are coordinates, and the tuple of arguments can be transformed by `array` to a NumPy array. Assume in all operations that the involved vectors have equal dimension (typically that `other` has the same dimension as `self`). Recall to return `Vec` objects from all arithmetic operations, not NumPy arrays, because the next operation with the vector will then not take place in `Vec` but in NumPy. If `self.v` is the attribute holding the vector as a NumPy array, the addition operator will typically be implemented as

```
class Vec:
    ...
    def __add__(self, other):
        return Vec(selv.v + other.v)
```

Name of program file: `Vec.py`. ◇

**Exercise 7.40.** *Use classes in the program from Ch. 6.6.2.*

Modify the `files/students.py` program described in Chapter 6.6.2 by making the values of the `data` dictionary instances of class `Student`. This class contains a student's name and a list of the courses. Each course is represented by an instance of class `Course`. This class contains the course name, the semester, the credit points, and the grade. Make

`__str__` and/or `__repr__` write out the contents of the objects. Name of program file: `Student_Course.py`. ◇

**Exercise 7.41.** *Use a class in Exer. 6.28.*

The purpose of this exercise is to make the program from Exercise 6.28 on page 333 more flexible by creating a class that runs and archives all the experiments. Here is a sketch of the class:

```
class GrowthLogistic:
    def __init__(self, show_plot_on_screen=False):
        self.experiments = []
        self.show_plot_on_screen = show_plot_on_screen
        self.remove_plot_files()

    def run_one(self, y0, q, N):
        """Run one experiment."""
        # compute y[n] in a loop...
        plotfile = 'tmp_y0_%g_q_%g_N_%d.png' % (y0, q, N)
        self.experiments.append({'y0': y0, 'q': q, 'N': N,
                                 'mean': mean(y[20:]),
                                 'y': y, 'plotfile': plotfile})
        # make plot...

    def run_many(self, y0_list, q_list, N):
        """Run many experiments."""
        for q in q_list:
            for y0 in y0_list:
                self.run_one(y0, q, N)

    def remove_plot_files(self):
        """Remove plot files with names tmp_y0*.png."""
        import os, glob
        for plotfile in glob.glob('tmp_y0*.png'):
            os.remove(plotfile)

    def report(self, filename='tmp.html'):
        """
        Generate an HTML report with plots of all
        experiments generated so far.
        """
        # open file and write HTML header...
        for e in self.experiments:
            html.write('<p><img src="%s">\n' % e['plotfile'])
        # write HTML footer and close file...
```

Each time the `run_one` method is called, data about the current experiment is stored in the `experiments` list. Note that `experiments` contains a list of dictionaries. When desired, we can call the `report` method to collect all the plots made so far in an HTML report. A typical use of the class goes as follows:

```
N = 50
g = GrowthLogistic()
g.run_many(y0_list=[0.01, 0.3],
           q_list=[0.1, 1, 1.5, 1.8] + [2, 2.5, 3], N=N)
g.run_one(y0=0.01, q=3, N=1000)
g.report()
```

Make a complete implementation of class `GrowthLogistic` and test it with the small program above. The program file should be constructed as a module. Name of program file: `growth_logistic5.py`. ◇

**Exercise 7.42.** *Apply the class from Exer. 7.41 interactively.*

Class `GrowthLogistic` from Exercise 7.41 is very well suited for interactive exploration. Here is a possible sample session for illustration:

```
>>> from growth_logistic5 import GrowthLogistic
>>> g = GrowthLogistic(show_plot_on_screen=True)
>>> q = 3
>>> g.run_one(0.01, q, 100)
>>> y = g.experiments[-1]['y']
>>> max(y)
1.3326056469620293
>>> min(y)
0.0029091569028512065
```

Extend this session with an investigation of the oscillations in the solution $y_n$. For this purpose, make a function for computing the local maximum values $y_n$ and the corresponding indices where these local maximum values occur. We can say that $y_i$ is a local maximum value if

$$y_{i-1} < y_i > y_{i+1}\,.$$

Plot the sequence of local maximum values in a new plot. If $I_0, I_1, I_2, \ldots$ constitute the set of increasing indices corresponding to the local maximum values, we can define the periods of the oscillations as $I_1 - I_0$, $I_2 - I_1$, and so forth. Plot the length of the periods in a separate plot. Repeat this investigation for $q = 2.5$. Log the whole session as explained in Exercise 1.11 (page 45). Name of program file: `GrowthLogistic_interactive.py`. ◇

**Exercise 7.43.** *Find the optimal production for a company.*

The company PROD produces two different products, $P_1$ and $P_2$, based on three different raw materials, $M_1$, $M_2$ og $M_3$. The following table shows how much of each raw material $M_i$ that is required to produce *a single unit* of each product $P_j$:

| | $P_1$ | $P_2$ |
|---|---|---|
| $M_1$ | 2 | 1 |
| $M_2$ | 5 | 3 |
| $M_3$ | 0 | 4 |

For instance, to produce one unit of $P_2$ one needs 1 unit of $M_1$, 3 units of $M_2$ and 4 units of $M_3$. Furthermore, PROD has available 100, 80 and 150 units of material $M_1$, $M_2$ and $M_3$ respectively (for the time period considered). The revenue per produced unit of product $P_1$ is 150 NOK, and for one unit of $P_2$ it is 175 NOK. On the other hand the raw materials $M_1$, $M_2$ and $M_3$ cost 10, 17 and 25 NOK per unit, respectively. The question is: How much should PROD produce of each product? We here assume that PROD wants to maximize its net revenue (which is revenue minus costs).

a) Let $x$ and $y$ be the number of units produced of product $P_1$ and $P_2$, respectively. Explain why the total revenue $f(x, y)$ is given by

$$f(x, y) = 150x - (10 \cdot 2 + 17 \cdot 5)x + 175y - (10 \cdot 1 + 17 \cdot 3 + 25 \cdot 4)y$$

and simplify this expression. The function $f(x, y)$ is *linear* in $x$ and $y$ (check that you know what linearity means).
b) Explain why PROD's problem may be stated mathematically as follows:

$$\begin{aligned} &\text{maximize} \quad f(x, y) \\ &\text{subject to} \\ &\qquad\qquad 2x + \ \ y \leq 100 \\ &\qquad\qquad 5x + 3y \leq 80 \\ &\qquad\qquad\qquad 4y \leq 150 \\ &\qquad\qquad x \geq 0, y \geq 0. \end{aligned} \tag{7.19}$$

This is an example of a *linear optimization problem.*
c) The production $(x, y)$ may be considered as a point in the plane. Illustrate geometrically the set $T$ of all such points that satisfy the constraints in model (7.19). Every point in this set is called a *feasible point.* (Hint: For every inequality determine first the straight line obtained by replacing the inequality by equality. Then, find the points satisfying the inequality (a halfplane), and finally, intersect these halfplanes.)
d) Make a program `optimization1.py` for drawing the straight lines defined by the inequalities. Each line can be written as $ax + by = c$. Let the program read each line from the command line as a list of the $a$, $b$, and $c$ values. In the present case the command-line arguments will be

```
'[2,1,100]' '[5,3,80]' '[0,4,150]' '[1,0,0]' '[0,1,0]'
```

(Hint: Perform an `value` on the elements of `sys.argv[1:]` to get $a$, $b$, and $c$ for each line as a list in the program.)
e) Let $\alpha$ be a positive number and consider the *level set* of the function $f$, defined as the set

$$L_\alpha = \{(x, y) \in T : f(x, y) = \alpha\}.$$

This set consists of all feasible points having the same net revenue $\alpha$. Extend the program with two new command-line arguments holding $p$ and $q$ for a function $f(x, y) = px + qy$. Use this information to compute the level set lines $y = \alpha/q - px/q$, and plot the level set lines for some different values of $\alpha$ (use the $\alpha$ value in the legend for each line).
f) Use what you saw in e) to solve the problem (7.19) geometrically. (Hint: How large can you choose $\alpha$ such that $L_\alpha$ is nonempty?) This solution is called an *optimal solution.*

Name of program file: `optimization1.py`. ◇

**Exercise 7.44.** *Extend the program from Exer. 7.43.*

Assume that we have other values on the revenues and costs than the actual numbers in Exercise 7.43. Explain why (7.19), with these new parameter values, still has an optimal solution lying in a corner point of $T$. Extend the program from Exercise 7.43 to calculate all the corner points of a region $T$ in the plane determined by the linear inequalities like those listed in Exercise 7.43. Moreover, the program shall compute the maximum of a given linear function $f(x, y) = px + qy$ over $T$ by calculating the function values in the corner points and finding the smallest function value. Name of program file: `optimization2.py`.

The example in Exercises 7.43 and 7.44 is from *linear optimization*, also called *linear programming*. Most universities and business schools have a good course in this important area of applied mathematics. ◇

**Exercise 7.45.** *Model the economy of fishing.*

A population of fish is governed by the differential equation

$$\frac{dx}{dt} = \frac{1}{10}x\left(1 - \frac{x}{100}\right) - h, \quad x(0) = 500, \tag{7.20}$$

where $x(t)$ is the size of the population at time $t$ and $h$ is the harvest.

a) Assume $h = 0$. Find an exact solution for $x(t)$. For which value of $t$ is $\frac{dx}{dt}$ largest? For which value of $t$ is $\frac{1}{x}\frac{dx}{dt}$ largest?
b) Solve the differential equation (7.20) by the Forward Euler method. Plot the numerical and exact solution in the same plot.
c) Suppose the harvest $h$ depends on the fishers' efforts, $E$, in the following way: $h = qxE$, with $q$ as a constant. Set $q = 0.1$ and assume $E$ is constant. Show the effect of $E$ on $x(t)$ by plotting several curves, corresponding to different $E$ values, in the same figure.
d) The fishers' total revenue is given by $\pi = ph - \frac{c}{2}E^2$, where $p$ is a constant. In the literature about the economy of fisheries, one is often interested in how a fishery will develop in the case the harvest is not regulated. Then new fishers will appear as long as there is money to earn ($\pi > 0$). It can (for simplicity) be reasonable to model the dependence of $E$ on $\pi$ as

$$\frac{dE}{dt} = \gamma\pi, \tag{7.21}$$

where $\gamma$ is a constant. Solve the system of differential equations for $x(t)$ and $E(t)$ by the Forward Euler method, and plot the curve with points $(x(t), E(t))$ in the two cases $\gamma = 1/2$ and $\gamma \to \infty$.

Name of program file: `fishery.py`. ◇

# 8 Random Numbers and Simple Games

Random numbers have many applications in science and computer programming, especially when there are significant uncertainties in a phenomenon of interest. The purpose of this chapter is to look at some practical problems involving random numbers and learn how to program with such numbers. We shall make several games and also look into how random numbers can be used in physics. You need to be familiar with the first four chapters in order to study the present chapter, but a few examples and exercises will require familiarity with the class concept from Chapter 7.

The key idea in computer simulations with random numbers is first to formulate an algorithmic description of the phenomenon we want to study. This description frequently maps directly onto a quite simple and short Python program, where we use random numbers to mimic the uncertain features of the phenomenon. The program needs to perform a large number of repeated calculations, and the final answers are "only" approximate, but the accuracy can usually be made good enough for practical purposes. Most programs related to the present chapter produce their results within a few seconds. In cases where the execution times become large, we can vectorize the code. Vectorized computations with random numbers is definitely the most demanding topic in this chapter, but is not mandatory for seeing the power of mathematical modeling via random numbers.

All files associated with the examples in this chapter are found in the folder `src/random`.

## 8.1 Drawing Random Numbers

Python has a module `random` for generating random numbers. The function call `random.random()` generates a random number in the half open interval[1] $[0, 1)$. We can try it out:

```
>>> import random
>>> random.random()
0.81550546885338104
>>> random.random()
0.44913326809029852
>>> random.random()
0.88320653116367454
```

All computations of random numbers are based on deterministic algorithms (see Exercise 8.16 for an example), so the sequence of numbers cannot be truly random. However, the sequence of numbers appears to lack any pattern, and we can therefore view the numbers as random[2].

### 8.1.1 The Seed

Every time we import `random`, the subsequent sequence of `random.random()` calls will yield different numbers. For debugging purposes it is useful to get the same sequence of random numbers every time we run the program. This functionality is obtained by setting a *seed* before we start generating numbers. With a given value of the seed, one and only one sequence of numbers is generated. The seed is an integer and set by the `random.seed` function:

```
>>> random.seed(121)
```

Let us generate two series of random numbers at once, using a list comprehension and a format with two decimals only:

```
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
>>> ['%.2f' % random.random() for i in range(7)]
['0.31', '0.61', '0.61', '0.58', '0.16', '0.43', '0.39']
```

If we set the seed to 2 again, the sequence of numbers is regenerated:

```
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
```

If we do not give a seed, the `random` module sets a seed based on the current time. That is, the seed will be different each time we run the

[1] In the half open interval $[0, 1)$ the lower limit is included, but the upper limit is not.

[2] What it means to view the numbers as random has fortunately a firm mathematical foundation, so don't let the fact that random numbers are deterministic stop you from using them.

program and consequently the sequence of random numbers will also be different from run to run. This is what we want in most applications. However, we recommend to always set a seed during program development to simplify debugging and verification.

### 8.1.2 Uniformly Distributed Random Numbers

The numbers generated by `random.random()` tend to be equally distributed between 0 and 1, which means that there is no part of the interval $[0, 1)$ with more random numbers than other parts. We say that the distribution of random numbers in this case is *uniform*. The function `random.uniform(a,b)` generates uniform random numbers in the half open interval $[a, b)$, where the user can specify $a$ and $b$. With the following program (in file `uniform_numbers0.py`) we may generate lots of random numbers in the interval $[-1, 1)$ and visualize how they are distributed :

```
import random
random.seed(42)
N = 500  # no of samples
x = range(N)
y = [random.uniform(-1,1) for i in x]
from scitools.easyviz import *
plot(x, y, '+', axis=[0,N-1,-1.2,1.2])
```

Figure 8.1 shows the values of these 500 numbers, and as seen, the numbers appear to be random and uniformly distributed between $-1$ and 1.

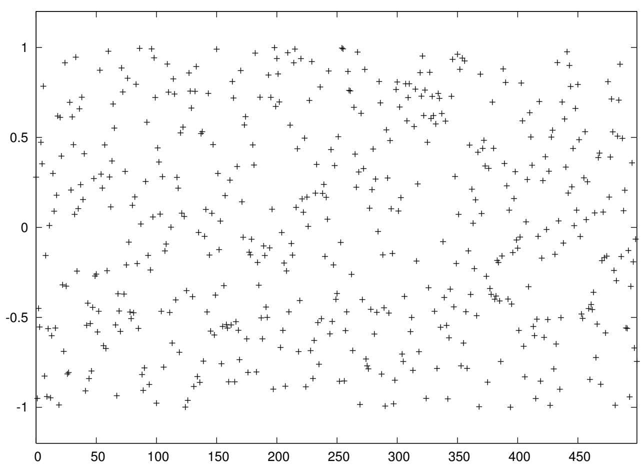



**Fig. 8.1** The values of 500 random numbers drawn from the uniform distribution on $[-1, 1)$.

### 8.1.3 Visualizing the Distribution

It is of interest to see how $N$ random numbers in an interval $[a, b]$ are distributed throughout the interval, especially as $N \to \infty$. For example, when drawing numbers from the uniform distribution, we expect that no parts of the interval get more numbers than others. To visualize the distribution, we can divide the interval into subintervals and display how many numbers there are in each subinterval.

Let us formulate this method more precisely. We divide the interval $[a, b)$ into $n$ equally sized subintervals, each of length $h = (b - a)/n$. These subintervals are called *bins*. We can then draw $N$ random numbers by calling `random.random()` $N$ times. Let $\hat{H}(i)$ be the number of random numbers that fall in bin no. $i$, $[a + ih, a + (i + 1)h]$, $i = 0, \ldots, n - 1$. If $N$ is small, the value of $\hat{H}(i)$ can be quite different for the different bins, but as $N$ grows, we expect that $\hat{H}(i)$ varies little with $i$.

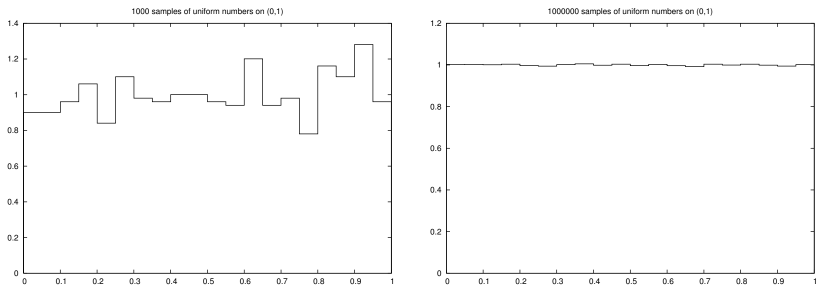
Ideally, we would be interested in how the random numbers are distributed as $N \to \infty$ and $n \to \infty$. One major disadvantage is that $\hat{H}(i)$ increases as $N$ increases, and it decreases with $n$. The quantity $\hat{H}(i)/N$, called the frequency count, will reach a finite limit as $N \to \infty$. However, $\hat{H}(i)/N$ will be smaller and smaller as we increase the number of bins. The quantity $H(i) = \hat{H}(i)/(Nh)$ reaches a finite limit as $N, n \to \infty$. The probability that a random number lies inside subinterval no. $i$ is then $\hat{H}(i)/N = H(i)h$.

We can visualize $H(i)$ as a bar diagram (see Figure 8.2), called a *normalized histogram*. We can also define a piecewise constant function $p(x)$ from $H(i)$: $p(x) = H(i)$ for $x \in [a+ih, a+(i+1)h)$, $i = 0, \ldots, n-1$. As $n, N \to \infty$, $p(x)$ approaches the probability density function of the distribution in question. For example, `random.uniform(a,b)` draws numbers from the uniform distribution on $[a, b)$, and the probability density function is constant, equal to $1/(b - a)$. As we increase $n$ and $N$, we therefore expect $p(x)$ to approach the constant $1/(b - a)$.

The function `compute_histogram` from `scitools.std` returns two arrays `x` and `y` such that `plot(x,y)` plots the piecewise constant function $p(x)$. The plot is hence the histogram of the set of random samples. The program below exemplifies the usage:

```
from scitools.std import plot, compute_histogram
import random
samples = [random.random() for i in range(100000)]
x, y = compute_histogram(samples, nbins=20)
plot(x, y)
```

Figure 8.2 shows two plots corresponding to $N$ taken as $10^3$ and $10^6$. For small $N$, we see that some intervals get more random numbers than others, but as $N$ grows, the distribution of the random numbers becomes more and more equal among the intervals. In the limit $N \to \infty$, $p(x) \to 1$, which is illustrated by the plot.




(a) (b)

**Fig. 8.2** The histogram of uniformly distributed random numbers in 20 bins.

### 8.1.4 Vectorized Drawing of Random Numbers

There is a `random` module in the Numerical Python package which can be used to efficiently draw a possibly large array of random numbers:

```
from numpy import random
r = random.random()            # one number between 0 and 1
r = random.random(size=10000)  # array with 10000 numbers
r = random.uniform(-1, 10)     # one number between -1 and 10
r = random.uniform(-1, 10, size=10000)  # array
```

There are thus two `random` modules to be aware of: one in the standard Python library and one in `numpy`. For drawing uniformly distributed numbers, the two `random` modules have the same interface, except that the functions from `numpy`'s `random` module has an extra `size` parameter. Both modules also have a `seed` function for fixing the seed.

Vectorized drawing of random numbers using `numpy`'s `random` module is efficient because all the numbers are drawn "at once" in fast C code. You can measure the efficiency gain with the `time.clock()` function as explained on page 447 and in Appendix E.6.1.

*Warning.* It is easy to do an `import random` followed by a `from scitools.std import *` or a `from numpy import *` without realizing that the latter two import statements import a name `random` that overwrites the same name that was imported in `import random`. The result is that the effective random module becomes the one from `numpy`. A possible solution to this problem is to introduce a different name for Python's `random` module:

```
import random as random_number
```

We will use this convention in the rest of the book. When you see only the word `random` you then know that this is `numpy.random`.

### 8.1.5 Computing the Mean and Standard Deviation

You probably know the formula for the mean or average of a set of $n$ numbers $x_0, x_1, \ldots, x_{n-1}$:

$$x_m = \frac{1}{n}\sum_{j=0}^{n-1} x_j \,. \tag{8.1}$$

The amount of spreading of the $x_i$ values around the mean $x_m$ can be measured by the *variance*[3],

$$x_v = \frac{1}{n}\sum_{j=0}^{n-1}(x_j - x_m)^2 \,. \tag{8.2}$$

A variant of this formula reads

$$x_v = \frac{1}{n}\left(\sum_{j=0}^{n-1} x_j^2\right) - x_m^2 \,. \tag{8.3}$$

The good thing with this latter formula is that one can, as a statistical experiment progresses and $n$ increases, record the sums

$$s_m = \sum_{j=0}^{q-1} x_j, \quad s_v = \sum_{j=0}^{q-1} x_j^2 \tag{8.4}$$

and then, when desired, efficiently compute the most recent estimate on the mean value and the variance after $q$ samples by

$$x_m = s_m/q, \quad x_v = s_v/q - s_m^2/q^2 \,. \tag{8.5}$$

The *standard deviation*

$$x_s = \sqrt{x_v} \tag{8.6}$$

is often used as an alternative to the variance, because the standard deviation has the same unit as the measurement itself. A common way to express an uncertain quantity $x$, based on a data set $x_0, \ldots, x_{n-1}$, from simulations or physical measurements, is $x_m \pm x_s$. This means that $x$ has an uncertainty of one standard deviation $x_s$ to either side of the mean value $x_m$. With probability theory and statistics one can provide many other, more precise measures of the uncertainty, but that is the topic of a different course.

Below is an example where we draw numbers from the uniform distribution on $[-1, 1)$ and compute the evolution of the mean and standard

[3] Textbooks in statistics teach you that it is more appropriate to divide by $n-1$ instead of $n$, but we are not going to worry about that fact in this book.

deviation 10 times during the experiment, using the formulas (8.1) and (8.3)–(8.6):

```
import sys
N = int(sys.argv[1])
import random as random_number
from math import sqrt
sm = 0; sv = 0
for q in range(1, N+1):
    x = random_number.uniform(-1, 1)
    sm += x
    sv += x**2

    # write out mean and st.dev. 10 times in this loop:
    if q % (N/10) == 0:
        xm = sm/q
        xs = sqrt(sv/q - xm**2)
        print '%10d mean: %12.5e  stdev: %12.5e' % (q, xm, xs)
```

The `if` test applies the *mod* function, `a % b` is 0 if `b` times an integer equals `a`. The particular `if` test here is true when `i` equals 0, `N/10`, `2*N/10`, ..., `N`, i.e., 10 times during the execution of the loop. The program is available in the file `mean_stdev_uniform1.py`. A run with $N = 10^6$ gives the output

```
    100000 mean:  1.86276e-03  stdev:  5.77101e-01
    200000 mean:  8.60276e-04  stdev:  5.77779e-01
    300000 mean:  7.71621e-04  stdev:  5.77753e-01
    400000 mean:  6.38626e-04  stdev:  5.77944e-01
    500000 mean: -1.19830e-04  stdev:  5.77752e-01
    600000 mean:  4.36091e-05  stdev:  5.77809e-01
    700000 mean: -1.45486e-04  stdev:  5.77623e-01
    800000 mean:  5.18499e-05  stdev:  5.77633e-01
    900000 mean:  3.85897e-05  stdev:  5.77574e-01
   1000000 mean: -1.44821e-05  stdev:  5.77616e-01
```

We see that the mean is getting smaller and approaching zero as expected since we generate numbers between $-1$ and 1. The theoretical value of the standard deviation, as $N \to \infty$, equals $\sqrt{1/3} \approx 0.57735$.

We have also made a corresponding vectorized version of the code above using `numpy.random` and the ready-made functions `numpy.mean`, `numpy.var`, and `numpy.std` for computing the mean, variance, and standard deviation (respectively) of an array of numbers:

```
import sys
N = int(sys.argv[1])
from numpy import random, mean, var, std, sqrt
x = random.uniform(-1, 1, size=N)
xm = mean(x)
xv = var(x)
xs = std(x)
print '%10d mean: %12.5e  stdev: %12.5e' % (N, xm, xs)
```

This program can be found in the file `mean_stdev_uniform2.py`.

### 8.1.6 The Gaussian or Normal Distribution

In some applications we want random numbers to cluster around a specific value $m$. This means that it is more probable to generate a

number close to $m$ than far away from $m$. A widely used distribution with this qualitative property is the Gaussian or normal distribution[4]. The normal distribution has two parameters: the mean value $m$ and the standard deviation $s$. The latter measures the width of the distribution, in the sense that a small $s$ makes it less likely to draw a number far from the mean value, and a large $s$ makes more likely to draw a number far from the mean value.

Single random numbers from the normal distribution can be generated by

```
import random as random_number
r = random_number.normalvariate(m, s)
```

while efficient generation of an array of length `N` is enabled by

```
from numpy import random
r = random.normal(m, s, size=N)
```

The following program draws `N` random numbers from the normal distribution in a loop, computes the mean and standard devation, and plots the histogram:

```
N = int(sys.argv[1])
m = float(sys.argv[2])
s = float(sys.argv[3])

import random as random_number
random_number.seed(12)   # for debugging/testing
from scitools.std import *

samples = [random_number.normalvariate(m, s) for i in range(N)]
x, y = compute_histogram(samples, 20, piecewise_constant=True)

print mean(samples), std(samples)
plot(x, y)
title('%d samples of Gaussian random numbers on (0,1)' % N)
hardcopy('tmp.eps')
```

The corresponding program file is `normal_numbers1.py`, which gives a mean of $-0.00253$ and a standard deviation of 0.99970 when run with `N` as 1 million, `m` as 0, and `s` equal to 1. Figure 8.3 shows that the random numbers cluster around the mean $m = 0$ in a histogram. This normalized histogram will, as `N` goes to infinity, approach a bell-shaped function, known as the normal distribution probability density function, given in (1.6) on page 45.

## 8.2 Drawing Integers

Suppose we want to draw a random integer among the values 1, 2, 3, and 4, and that each of the four values is equally probable. One

[4] For example, the blood pressure among adults of one gender has values that follow a normal distribution.

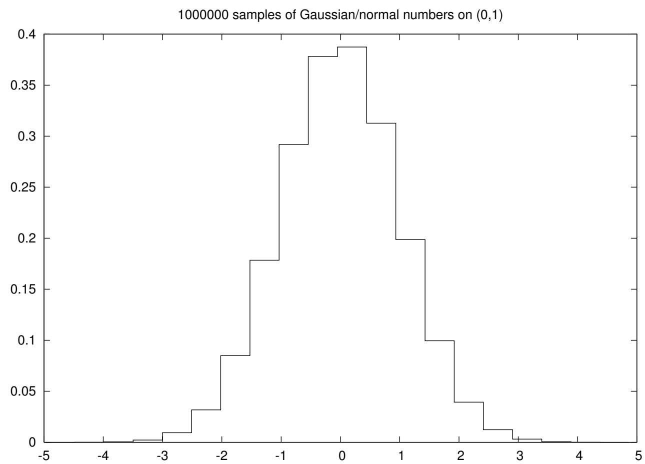



**Fig. 8.3** Normalized histogram of 1 million random numbers drawn from the normal distribution.

possibility is to draw real numbers from the uniform distribution on, e.g., $[0, 1)$ and divide this interval into four equal subintervals:

```
import random as random_number
r = random_number.random()
if 0 <=  r < 0.25:
    r = 1
elif 0.25 <= r < 0.5:
    r = 2
elif 0.5 <= r < 0.75:
    r = 3
else:
    r = 4
```

Nevertheless, the need for drawing uniformly distributed integers occurs quite frequently, so there are special functions for returning random integers in a specified interval $[a, b]$.

### 8.2.1 Random Integer Functions

Python's `random` module has a built-in function `randint(a,b)` for drawing an integer in $[a, b]$, i.e., the return value is among the numbers `a`, `a+1`, ..., `b-1`, `b`.

```
import random as random_number
r = random_number.randint(a, b)
```

The `numpy.random.randint(a, b, N)` function has a similar functionality for vectorized drawing of an array of length `N` of random integers in $[a, b)$. The upper limit `b` is not among the drawn numbers, so if we want to draw from `a`, `a+1`, ..., `b-1`, `b`, we must write

```
from numpy inport random
r = random.randint(a, b+1, N)
```

Another function, `random_integers(a, b, N)`, also in `numpy.random`, includes the upper limit `b` in the possible set of random integers:

```
from numpy inport random
r = random.random_integers(a, b, N)
```

### 8.2.2 Example: Throwing a Die

We can make a program that lets the computer throw a die `N` times and count how many times we get six eyes:

```
import random as random_number
import sys
N = int(sys.argv[1])   # perform N experiments
M = 0                  # no of times we get 6 eyes
for i in xrange(N):
    outcome = random_number.randint(1, 6)
    if outcome == 6:
        M += 1
print 'Got six %d times out of %d' % (M, N)
```

We use `xrange` instead of `range` because the former is more efficient when `N` is large (see remark in Exercise 2.46). The vectorized version of this code can be expressed as follows:

```
from numpy import random, sum
import sys
N = int(sys.argv[1])
eyes = random.randint(1, 7, N)
success = eyes == 6  # True/False array
M = sum(success)     # treats True as 1, False as 0
print 'Got six %d times out of %d' % (M, N)
```

The `eyes == 6` construction results in an array with `True` or `False` values, and `sum` applied to this array treats `True` as 1 and `False` as 0 (the integer equivalents to the boolean values), so the sum is the number of elements in `eyes` that equals 6. A very important point here for computational efficiency is to use `sum` from `numpy` and not the standard `sum` function that is available in standard Python. With the former `sum` function, the vectorized version runs about 50 times faster than the scalar version. (With the standard `sum` function in Python, the vectorized versions is in fact slower than the scalar version.)

The two small programs above are found in the files `roll_die.py` and `roll_die_vec.py`, respectively. You can try the programs and see how much faster the vectorized version is (`N` probably needs to be of size at least $10^6$ to see any noticable differences for practical purposes).

### 8.2.3 Drawing a Random Element from a List

Given a list `a`, the statement

```
re = random_number.choice(a)
```

picks out an element of `a` at random, and `re` refers to this element. The shown call to `random_number.choice` is the same as

```
re = a[random_number.randint(0, len(a)-1)]
```

There is also a function `shuffle` that permutates the list elements in a random order:

```
random_number.shuffle(a)
```

Picking now `a[0]`, for instance, has the same effect as `random.choice` on the original, unshuffled list. Note that `shuffle` changes the list given as argument.

The `numpy.random` module has also a `shuffle` function with the same functionality.

A small session illustrates the various methods for picking a random element from a list:

```
>>> awards = ['car', 'computer', 'ball', 'pen']
>>> import random as random_number
>>> random_number.choice(awards)
'car'
>>> awards[random_number.randint(0, len(awards)-1)]
'pen'
>>> random_number.shuffle(awards)
>>> awards[0]
'computer'
```

### 8.2.4 Example: Drawing Cards from a Deck

The following function creates a deck of cards, where each card is represented as a string, and the deck is a list of such strings:

```
def make_deck():
    ranks = ['A', '2', '3', '4', '5', '6', '7',
             '8', '9', '10', 'J', 'Q', 'K']
    suits = ['C', 'D', 'H', 'S']
    deck = []
    for s in suits:
        for r in ranks:
            deck.append(s + r)
    random_number.shuffle(deck)
    return deck
```

Here, `'A'` means an ace, `'J'` represents a jack, `'Q'` represents a queen, `'K'` represents a king, `'C'` stands for clubs, `'D'` stands for diamonds, `'H'` means hearts, and `'S'` means spades. The computation of the list

deck can alternatively (and more compactly) be done by a one-line list comprehension:

```
    deck = [s+r for s in suits for r in ranks]
```

We can draw a card at random by

```
deck = make_deck()
card = deck[0]
del deck[0]
# or better:
card = deck.pop(0)  # return and remove element with index 0
```

Drawing a hand of n cards from a shuffled deck is accomplished by

```
def deal_hand(n, deck):
    hand = [deck[i] for i in range(n)]
    del deck[:n]
    return hand, deck
```

Note that we must return deck to the calling code since this list is changed. Also note that the n first cards of the deck are random cards if the deck is shuffled (and any deck made by make_deck is shuffled).

The following function deals cards to a set of players:

```
def deal(cards_per_hand, no_of_players):
    deck = make_deck()
    hands = []
    for i in range(no_of_players):
        hand, deck = deal_hand(cards_per_hand, deck)
        hands.append(hand)
    return hands
```

```
players = deal(5, 4)
import pprint; pprint.pprint(players)
```

The players list may look like

```
    [['D4', 'CQ', 'H10', 'DK', 'CK'],
     ['D7', 'D6', 'SJ', 'S4', 'C5'],
     ['C3', 'DQ', 'S3', 'C9', 'DJ'],
     ['H6', 'H9', 'C6', 'D5', 'S6']]
```

The next step is to analyze a hand. Of particular interest is the number of pairs, three of a kind, four of a kind, etc. That is, how many combinations there are of n_of_a_kind cards of the same rank (e.g., n_of_a_kind=2 finds the number of pairs):

```
def same_rank(hand, n_of_a_kind):
    ranks = [card[1:] for card in hand]
    counter = 0
    already_counted = []
    for rank in ranks:
        if rank not in already_counted and \
                ranks.count(rank) == n_of_a_kind:
            counter += 1
            already_counted.append(rank)
    return counter
```

Note how convenient the `count` method in list objects is for counting how many copies there are of one element in the list.

Another analysis of the hand is to count how many cards there are of each suit. A dictionary with the suit as key and the number of cards with that suit as value, seems appropriate to return. We pay attention only to suits that occur more than once:

```
def same_suit(hand):
    suits = [card[0] for card in hand]
    counter = {}   # counter[suit] = how many cards of suit
    for suit in suits:
        count = suits.count(suit)
        if count > 1:
            counter[suit] = count
    return counter
```

For a set of players we can now analyze their hands:

```
for hand in players:
    print """\
The hand %s
    has %d pairs, %s 3-of-a-kind and
    %s cards of the same suit.""" % \
    (', '.join(hand), same_rank(hand, 2),
     same_rank(hand, 3),
     '+'.join([str(s) for s in same_suit(hand).values()]))
```

The values we feed into the printf string undergo some massage: we join the card values with comma and put a plus in between the counts of cards with the same suit. (The `join` function requires a string argument. That is why the integer counters of cards with the same suit, returned from `same_suit`, must be converted to strings.) The output of the `for` loop becomes

```
The hand D4, CQ, H10, DK, CK
    has 1 pairs, 0 3-of-a-kind and
    2+2 cards of the same suit.
The hand D7, D6, SJ, S4, C5
    has 0 pairs, 0 3-of-a-kind and
    2+2 cards of the same suit.
The hand C3, DQ, S3, C9, DJ
    has 1 pairs, 0 3-of-a-kind and
    2+2 cards of the same suit.
The hand H6, H9, C6, D5, S6
    has 0 pairs, 1 3-of-a-kind and
    2 cards of the same suit.
```

The file `cards.py` contains the functions `make_deck`, `hand`, `hand2`, `same_rank`, `same_suit`, and the test snippets above. With the `cards.py` file one can start to implement real card games.

### 8.2.5 Example: Class Implementation of a Deck

To work with a deck of cards with the code from the previous section one needs to shuffle a global variable `deck` in and out of functions. A set of functions that update global variables (like `deck`) is a primary candidate for a class: The global variables are stored as attributes and

the functions become class methods. This means that the code from the previous section is better implemented as a class. We introduce class `Deck` with a list of cards, `deck`, as attribute, and methods for dealing one or several hands and for putting back a card:

```
class Deck:
    def __init__(self):
        ranks = ['A', '2', '3', '4', '5', '6', '7',
                 '8', '9', '10', 'J', 'Q', 'K']
        suits = ['C', 'D', 'H', 'S']
        self.deck = [s+r for s in suits for r in ranks]
        random_number.shuffle(self.deck)

    def hand(self, n=1):
        """Deal n cards. Return hand as list."""
        hand = [self.deck[i] for i in range(n)]  # pick cards
        del self.deck[:n]                        # remove cards
        return hand

    def deal(self, cards_per_hand, no_of_players):
        """Deal no_of_players hands. Return list of lists."""
        return [self.hand(cards_per_hand) \
                for i in range(no_of_players)]

    def putback(self, card):
        """Put back a card under the rest."""
        self.deck.append(card)

    def __str__(self):
        return str(self.deck)
```

This class is found in the module file `Deck.py`. Dealing a hand of five cards to `p` players is coded as

```
from Deck import Deck
deck = Deck()
print deck
players = deck.deal(5, 4)
```

Here, `players` become a nested list as shown in Chapter 8.2.4.

One can go a step further and make more classes for assisting card games. For example, a card has so far been represented by a plain string, but we may well put that string in a class `Card`:

```
class Card:
    """Representation of a card as a string (suit+rank)."""
    def __init__(self, suit, rank):
        self.card = suit + str(rank)

    def __str__(self):   return self.card
    def __repr__(self):  return str(self)
```

Note that `str(self)` is equivalent to `self.__str__()`.

A `Hand` contains a set of `Card` instances and is another natural abstraction, and hence a candidate for a class:

```
class Hand:
    """Representation of a hand as a list of Card objects."""
    def __init__(self, list_of_cards):
        self.hand = list_of_cards
```

```
    def __str__(self):   return str(self.hand)
    def __repr__(self):  return str(self)
```

With the aid of classes `Card` and `Hand`, class `Deck` can be reimplemented as

```
class Deck:
    """Representation of a deck as a list of Card objects."""

    def __init__(self):
        ranks = ['A', '2', '3', '4', '5', '6', '7',
                 '8', '9', '10', 'J', 'Q', 'K']
        suits = ['C', 'D', 'H', 'S']
        self.deck = [Card(s,r) for s in suits for r in ranks]
        random_number.shuffle(self.deck)

    def hand(self, n=1):
        """Deal n cards. Return hand as a Hand object."""
        hand = Hand([self.deck[i] for i in range(n)])
        del self.deck[:n]         # remove cards
        return hand

    def deal(self, cards_per_hand, no_of_players):
        """Deal no_of_players hands. Return list of Hand obj."""
        return [self.hand(cards_per_hand) \
                for i in range(no_of_players)]

    def putback(self, card):
        """Put back a card under the rest."""
        self.deck.append(card)

    def __str__(self):
        return str(self.deck)

    def __repr__(self):
        return str(self)

    def __len__(self):
        return len(self.deck)
```

The module file `Deck2.py` contains this implementation. The usage of the two `Deck` classes is the same,

```
from Deck2 import Deck
deck = Deck()
players = deck.deal(5, 4)
```

with the exception that `players` in the last case holds a list of `Hand` instances, and each `Hand` instance holds a list of `Card` instances.

We stated in Chapter 7.3.9 that the `__repr__` method should return a string such that one can recreate the object from this string by the aid of `eval`. However, we did not follow this rule in the implementation of classes `Card`, `Hand`, and `Deck`. Why? The reason is that we want to print a `Deck` instance. Python's `print` or `pprint` on a list applies `repr(e)` to print an element `e` in the list. Therefore, if we had implemented

```
class Card:
    ...
    def __repr__(self):
        return "Card('%s', %s)" % (self.card[0], self.card[1:])
```

```
class Hand:
    ...
    def __repr__(self):  return 'Hand(%s)' % repr(self.hand)
```

a plain printing of the deck list of Hand instances would lead to output like

```
[Hand([Card('C', '10'), Card('C', '4'), Card('H', 'K'), ...]),
 ...,
 Hand([Card('D', '7'), Card('C', '5'), ..., Card('D', '9')])]
```

This output is harder to read than

```
[[C10, C4, HK, DQ, HQ],
 [SA, S8, H3, H10, C2],
 [HJ, C7, S2, CQ, DK],
 [D7, C5, DJ, S3, D9]]
```

That is why we let `__repr__` in classes `Card` and `Hand` return the same "pretty print" string as `__str__`, obtained by returning `str(self)`.

## 8.3 Computing Probabilities

With the mathematical rules from *probability theory* one may compute the probability that a certain event happens, say the probability that you get one black ball when drawing three balls from a hat with four black balls, six white balls, and three green balls. Unfortunately, theoretical calculations of probabilities may soon become hard or impossible if the problem is slightly changed. There is a simple "numerical way" of computing probabilities that is generally applicable to problems with uncertainty. The principal ideas of this approximate technique is explained below, followed by with three examples of increasing complexity.

### 8.3.1 Principles of Monte Carlo Simulation

Assume that we perform $N$ experiments where the outcome of each experiment is random. Suppose that some event takes place $M$ times in these $N$ experiments. An estimate of the probability of the event is then $M/N$. The estimate becomes more accurate as $N$ is increased, and the exact probability is assumed to be reached in the limit as $N \to \infty$. (Note that in this limit, $M \to \infty$ too, so for rare events, where $M$ may be small in a program, one must increase $N$ such that $M$ is sufficiently large for $M/N$ to become a good approximation to the probability.)

Programs that run a large number of experiments and record the outcome of events are often called *simulation programs*[5]. The mathematical technique of letting the computer perform lots of experiments

[5] This term is also applied for programs that solve equations arising in mathematical models in general, but it is particularly common to use the term when random numbers are used to estimate probabilities.

based on drawing random numbers is commonly called *Monte Carlo simulation*. This technique has proven to be extremely useful throughout science and industry in problems where there is uncertain or random behavior is involved[6]. For example, in finance the stock market has a random variation that must be taken into account when trying to optimize investments. In offshore engineering, environmental loads from wind, currents, and waves show random behavior. In nuclear and particle physics, random behavior is fundamental according to quantum mechanics and statistical physics. Many probabilistic problems can be calculated exactly by mathematics from probability theory, but very often Monte Carlo simulation is the only way to solve statistical problems. Chapters 8.3.2–8.3.4 applies examples to explain the essence of Monte Carlo simulation in problems with inherent uncertainty. However, also deterministic problems, such as integration of functions, can be computed by Monte Carlo simulation (see Chapter 8.5).

### 8.3.2 Example: Throwing Dice

What is the probability of getting at least six eyes twice when rolling four dice? The experiment is to roll four dice, and the event we are looking for appears when we get two or more dice with six eyes. A program `roll_dice1.py` simulating `N` such experiments may look like this:

```
import random as random_number
import sys
N = int(sys.argv[1])  # no of experiments
M = 0                 # no of successful events
for i in range(N):
    six = 0           # count the no of dice with a six
    r1 = random_number.randint(1, 6)
    if r1 == 6:
        six += 1
    r2 = random_number.randint(1, 6)
    if r2 == 6:
        six += 1
    r3 = random_number.randint(1, 6)
    if r3 == 6:
        six += 1
    r4 = random_number.randint(1, 6)
    if r4 == 6:
        six += 1
    # successful event?
    if six >= 2:
        M += 1
p = float(M)/N
print 'probability:', p
```

*Generalization.* We can easily parameterize how many dice (`ndice`) we roll in each experiment and how many dice with six eyes we want to see

[6] "As far as the laws of mathematics refer to reality, they are not certain, as far as they are certain, they do not refer to reality." –Albert Einstein, physicist, 1879-1955.

(`nsix`). Thereby, we get a shorter and more general code. The increased generality usually makes it easier to apply or adapt to new problems. The resulting program is found in `roll_dice2.py` and is listed below:

```
import random as random_number
import sys
N = int(sys.argv[1])      # no of experiments
ndice = int(sys.argv[2])  # no of dice
nsix = int(sys.argv[3])   # wanted no of dice with six eyes
M = 0                     # no of successful events
for i in range(N):
    six = 0               # how many dice with six eyes?
    for j in range(ndice):
        # roll die no. j:
        r = random_number.randint(1, 6)
        if r == 6:
            six += 1
    # successful event?
    if six >= nsix:
        M += 1
p = float(M)/N
print 'probability:', p
```

With this program we may easily change the problem setting and ask for the probability that we get six eyes $q$ times when we roll $q$ dice. The theoretical probability can be calculated to be $6^{-4} \approx 0.00077$, and a program performing $10^5$ experiments estimates this probability to 0.0008. For such small probabilities the number of successful events $M$ is small, and $M/N$ will not be a good approximation to the probability unless $M$ is reasonably large, which requires a very large $N$. The `roll_dice2.py` program runs quite slowly for one million experiments, so it is a good idea to try to vectorize the code to speed up the experiments. Unfortunately, this may constitute a challenge for newcomers to programming, as shown below.

*Vectorization.* In a vectorized version of the `roll_dice2.py` program, we generate a two-dimensional array of random numbers where the first dimension reflects the experiments and the second dimension reflects the trials in each experiment:

```
from numpy import random, sum
eyes = random.randint(1, 7, (N, ndice))
```

The next step is to count the number of successes in each experiment. For this purpose, we must avoid explicit loops if we want the program to run fast. In the present example, we can compare all rolls with 6, resulting in an array `compare` (dimension as `eyes`) with ones for rolls with 6 and 0 otherwise. Summing up the rows in `compare`, we are interested in the rows where the sum is equal to or greater than `nsix`. The number of such rows equals the number of successful events,

which we must divide by the total number of experiments to get the probability[7]:

```
compare = eyes == 6
nthrows_with_6 = sum(compare, axis=1)  # sum over columns (index 1)
nsuccesses = nthrows_with_6 >= nsix
M = sum(nsuccesses)
p = float(M)/N
```

The complete program is found in the file `roll_dice2_vec.py`. Getting rid of the two loops, as we obtained in the vectorized version, speeds up the probability estimation with a factor of 40. However, the vectorization is highly non-trivial, and the technique depends on details of how we define success of an event in an experiment.

### 8.3.3 Example: Drawing Balls from a Hat

Suppose there are 12 balls in a hat: four black, four red, and four blue. We want to make a program that draws three balls at random from the hat. It is natural to represent the collection of balls as a list. Each list element can be an integer 1, 2, or 3, since we have three different types of balls, but it would be easier to work with the program if the balls could have a color instead of an integer number. This is easily accomplished by defining color names:

```
colors = 'black', 'red', 'blue'   # (tuple of strings)
hat = []
for color in colors:
    for i in range(4):
        hat.append(color)
```

Drawing a ball at random is performed by

```
import random as random_number
color = random_number.choice(hat)
print color
```

Drawing $n$ balls without replacing the drawn balls requires us to remove an element from the hat when it is drawn. There are three ways to implement the procedure: (i) we perform a `hat.remove(color)`, (ii) we draw a random index with `randint` from the set of legal indices in the `hat` list, and then we do a `del hat[index]` to remove the element, or (iii) we can compress the code in (ii) to `hat.pop(index)`.

```
def draw_ball(hat):
    color = random_number.choice(hat)
    hat.remove(color)
```

[7] This code is considered advanced so don't be surprised if you dont't understand much of it. A first step toward understanding is to type in the code and write out the individual arrays for (say) $N = 2$. The use of `numpy`'s `sum` function is essential for efficiency.

```
    return color, hat

def draw_ball(hat):
    index = random_number.randint(0, len(hat)-1)
    color = hat[index]
    del hat[index]
    return color, hat

def draw_ball(hat):
    index = random_number.randint(0, len(hat)-1)
    color = hat.pop(index)
    return color, hat

# draw n balls from the hat:
balls = []
for i in range(n):
    color, hat = draw_ball(hat)
    balls.append(color)
print 'Got the balls', balls
```

We can extend the experiment above and ask the question: What is the probability of drawing two or more black balls from a hat with 12 balls, four black, four red, and four blue? To this end, we perform $N$ experiments, count how many times $M$ we get two or more black balls, and estimate the probability as $M/N$. Each experiment consists of making the `hat` list, drawing a number of balls, and counting how many black balls we got. The latter task is easy with the `count` method in list objects: `hat.count('black')` counts how many elements with value `'black'` we have in the list `hat`. A complete program for this task is listed below. The program appears in the file `balls_in_hat.py`.

```
import random as random_number

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random_number.randint(0, len(hat)-1)
    color = hat.pop(index)
    return color, hat

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random_number.randint(0, len(hat)-1)
    color = hat[index]
    del hat[index]
    return color, hat

def draw_ball(hat):
    """Draw a ball using list element."""
    color = random_number.choice(hat)
    hat.remove(color)
    return color, hat

def new_hat():
    colors = 'black', 'red', 'blue'   # (tuple of strings)
    hat = []
    for color in colors:
        for i in range(4):
            hat.append(color)
    return hat

n = int(raw_input('How many balls are to be drawn? '))
N = int(raw_input('How many experiments? '))
```

```
# run experiments:
M = 0  # no of successes
for e in range(N):
    hat = new_hat()
    balls = []           # the n balls we draw
    for i in range(n):
        color, hat = draw_ball(hat)
        balls.append(color)
    if balls.count('black') >= 2:  # at least two black balls?
        M += 1
print 'Probability:', float(M)/N
```

Running the program with $n = 5$ (drawing 5 balls each time) and $N = 4000$ gives a probability of 0.57. Drawing only 2 balls at a time reduces the probability to about 0.09.

One can with the aid of probability theory derive theoretical expressions for such probabilities, but it is much simpler to let the computer perform a large number of experiments to estimate an approximate probability.

A class version of the code in this section is better than the code presented, because we avoid shuffling the `hat` variable in and out of functions. Exercise 8.17 asks you to design and implement a class `Hat`.

### 8.3.4 Example: Policies for Limiting Population Growth

China has for many years officially allowed only one child per couple. However, the success of the policy has been somewhat limited. One challenge is the current overrepresentation of males in the population (families have favored sons to live up). An alternative policy is to allow each couple to continue getting children until they get a son. We can simulate both policies and see how a population will develop under the "one child" and the "one son" policies. Since we expect to work with a large population over several generations, we aim at vectorized code at once.

Suppose we have a collection of `n` individuals, called `parents`, consisting of males and females randomly drawn such that a certain portion (`male_portion`) constitutes males. The `parents` array holds integer values, 1 for male and 2 for females. We can introduce constants, `MALE=1` and `FEMALE=2`, to make the code easier to read. Our task is to see how the `parents` array develop from one generation to the next under the two policies. Let us first show how to draw the random integer array `parents` where there is a probability `male_portion` of getting the value `MALE`:

```
r = random.random(n)
parents = zeros(n, int)
MALE = 1; FEMALE = 2
parents[r <  male_portion] = MALE
parents[r >= male_portion] = FEMALE
```

The number of potential couples is the minimum of males and females. However, only a fraction (`fertility`) of the couples will actually get a child. Under the perfect "one child" policy, these couples can have one child each:

```
males = len(parents[parents==MALE])
females = len(parents) - males
couples = min(males, females)
n = int(fertility*couples)  # couples that get a child

# the next generation, one child per couple:
r = random.random(n)
children = zeros(n, int)
children[r <  male_portion] = MALE
children[r >= male_portion] = FEMALE
```

The code for generating a new population will be needed in every generation. Therefore, it is natural to collect the last statements statements in a separate function such that we can repeat the statements when needed.

```
def get_children(n, male_portion, fertility):
    n = int(fertility*n)
    r = random.random(n)
    children = zeros(n, int)
    children[r <  male_portion] = MALE
    children[r >= male_portion] = FEMALE
    return children
```

Under the "one son" policy, the families can continue getting a new child until they get the first son:

```
# first try:
children = get_children(couples, male_portion, fertility)

# continue with getting a new child for each daughter:
daughters = children[children == FEMALE]
while len(daughters) > 0:
    new_children = get_children(len(daughters),
                                male_portion, fertility)
    children = concatenate((children, new_children))
    daughters = new_children[new_children == FEMALE]
```

The program `birth_policy.py` organizes the code segments above for the two policies into a function `advance_generation`, which we can call repeatedly to see the evolution of the population.

```
def advance_generation(parents, policy='one child',
                       male_portion=0.5, fertility=1.0):
    males = len(parents[parents==MALE])
    females = len(parents) - males
    couples = min(males, females)

    if policy == 'one child':
        children = get_children(couples, male_portion, fertility)
    elif policy == 'one son':
        # first try at getting a child:
        children = get_children(couples, male_portion, fertility)
        # continue with getting a new child for each daughter:
```

```
        daughters = children[children == FEMALE]
        while len(daughters) > 0:
            new_children = get_children(len(daughters),
                                        male_portion, fertility)
            children = concatenate((children, new_children))
            daughters = new_children[new_children == FEMALE]
    return children
```

The simulation is then a matter of repeated calls to `advance_generation`:

```
N = 1000000              # population size
male_portion = 0.51
fertility = 0.92
# start with a "perfect" generation of parents:
parents = get_children(N, male_portion=0.5, fertility=1.0)
print 'one son policy, start: %d' % len(parents)
for i in range(10):
    parents = advance_generation(parents, 'one son',
                                 male_portion, fertility)
    print '%3d: %d' % (i+1, len(parents))
```

Under ideal conditions with unit `fertility` and a `male_portion` of 0.5, the program predicts that the "one child" policy halves the population from one generation to the next, while the "one son" policy, where we expect each couple to get one daughter and one son on average, keeps the population constant. Increasing `male_portion` slightly and decreasing `fertility`, which corresponds more to reality, will in both cases lead to a reduction of the population. You can try the program out with various values of these input parameters.

An obvious extension is to incorporate the effect that a portion of the population does not follow the policy and get $c$ children on average. The program `birth_policy.py` can account for the effect, which is quite dramatic: If 1% of the population does not follow the "one son" policy and get 4 children on average, the population grows with 50% over 10 generations (`male_portion` and `fertility` kept at the ideal values 0.5 and 1, respectively).

Normally, simple models like the difference equations (5.9) and (5.12), or the differential equations (B.11) or (B.23), are used to model population growth. However, these models track the number of individuals through time with a very simple growth factor from one generation to the next. The model above tracks each individual in the population and applies rules involving random actions to each individual. Such a detailed and much more computer-time consuming model can be used to see the effect of different policies. Using the results of this detailed model, we can (sometimes) estimate growth factors for simpler models so that these mimic the overall effect on the population size. Exercise 8.24 asks you to investigate if a certain realization of the "one son" policy leads to simple exponential growth.

## 8.4 Simple Games

This section presents the implementation of some simple games based on drawing random numbers. The games can be played by two humans, but here we consider a human versus the computer.

### 8.4.1 Guessing a Number

*The Game.* The computer determines a secret number, and the player shall guess the number. For each guess, the computer tells if the number is too high or too low.

*The Implementation.* We let the computer draw a random integer in an interval known to the player, let us say $[1, 100]$. In a `while` loop the program prompts the player for a guess, reads the guess, and checks if the guess is higher or lower than the drawn number. An appropriate message is written to the screen. We think the algorithm can be expressed directly as executable Python code:

```
import random as random_number
number = random_number.randint(1, 100)
attempts = 0  # count no of attempts to guess the number
guess = 0
while guess != number:
    guess = eval(raw_input('Guess a number: '))
    attempts += 1
    if guess == number:
        print 'Correct! You used', attempts, 'attempts!'
        break
    elif guess < number:
        print 'Go higher!'
    else:
        print 'Go lower!'
```

The program is available as the file `guessnumber.py`. Try it out! Can you come up with a strategy for reducing the number of attempts? See Exercise 8.25 for an automatic investigation of two possible strategies.

### 8.4.2 Rolling Two Dice

*The Game.* The player is supposed to roll two dice, and on beforehand guess the sum of the eyes. If the guess on the sum is $n$ and it turns out to be right, the player earns $n$ euros. Otherwise, the player must pay 1 euro. The machine plays in the same way, but the machine's guess of the number of eyes is a uniformly distributed number between 2 and 12. The player determines the number of rounds, $r$, to play, and receives $r$ euros as initial capital. The winner is the one that has the largest amount of euros after $r$ rounds, or the one that avoids to lose all the money.

*The Implementation.* There are three actions that we can naturally implement as functions: (i) roll two dice and compute the sum; (ii) ask the player to guess the number of eyes; (iii) draw the computer's guess of the number of eyes. One soon realizes that it is as easy to implement this game for an arbitrary number of dice as it is for two dice. Consequently we can introduce `ndice` as the number of dice. The three functions take the following forms:

```
import random as random_number

def roll_dice_and_compute_sum(ndice):
    return sum([random_number.randint(1, 6) \
                for i in range(ndice)])

def computer_guess(ndice):
    return random_number.randint(ndice, 6*ndice)

def player_guess(ndice):
    return input('Guess the sum of the no of eyes '\
                 'in the next throw: ')
```

We can now implement one round in the game for the player or the computer. The round starts with a capital, a guess is performed by calling the right function for guessing, and the capital is updated:

```
def play_one_round(ndice, capital, guess_function):
    guess = guess_function(ndice)
    throw = roll_dice_and_compute_sum(ndice)
    if guess == throw:
        capital += guess
    else:
        capital -= 1
    return capital, throw, guess
```

Here, `guess_function` is either `computer_guess` or `player_guess`.

With the `play_one_round` function we can run a number of rounds involving both players:

```
def play(nrounds, ndice=2):
    # start capital:
    player_capital = computer_capital = nrounds

    for i in range(nrounds):
        player_capital, throw, guess = \
             play_one_round(ndice, player_capital, player_guess)
        print 'YOU guessed %d, got %d' % (guess, throw)
        if player_capital == 0:
            print 'Machine won!'; sys.exit(0)

        computer_capital, throw, guess = \
            play_one_round(ndice, computer_capital, computer_guess)

        print 'Machine guessed %d, got %d' % (guess, throw)
        if computer_capital == 0:
            print = 'You won!'; sys.exit(0)

        print 'Status: you have %d euros, machine has %d euros' % \
              (player_capital, computer_capital)

    if computer_capital > player_capital:
```

```
        winner = 'Machine'
    else:
        winner = 'You'
    print winner, 'won!'
```

The name of the program is `ndice.py`.

*Example.* Here is a session (with a fixed seed of 20):

```
Guess the sum of the no of eyes in the next throw: 7
YOU guessed 7, got 11
Machine guessed 10, got 8
Status: you have 9 euros, machine has 9 euros

Guess the sum of the no of eyes in the next throw: 9
YOU guessed 9, got 10
Machine guessed 11, got 6
Status: you have 8 euros, machine has 8 euros

Guess the sum of the no of eyes in the next throw: 9
YOU guessed 9, got 9
Machine guessed 3, got 8
Status: you have 17 euros, machine has 7 euros
```

Exercise 8.27 asks you to perform simulations to determine whether a certain strategy can make the player win over the computer in the long run.

*A Class Version.* We can cast the previous code segment in a class. Many will argue that a class-based implementation is closer to the problem being modeled and hence easier to modify or extend.

A natural class is `Dice`, which can throw `n` dice:

```
class Dice:
    def __init__(self, n=1):
        self.n = n   # no of dice

    def throw(self):
        return [random_number.randint(1,6) \
                for i in range(self.n)]
```

Another natural class is `Player`, which can perform the actions of a player. Functions can then make use of `Player` to set up a game. A `Player` has a name, an initial captial, a set of dice, and a `Dice` object to throw the object:

```
class Player:
    def __init__(self, name, capital, guess_function, ndice):
        self.name = name
        self.capital = capital
        self.guess_function = guess_function
        self.dice = Dice(ndice)

    def play_one_round(self):
        self.guess = self.guess_function(self.dice.n)
        self.throw = sum(self.dice.throw())
        if self.guess == self.throw:
            self.capital += self.guess
        else:
            self.capital -= 1
        self.message()
        self.broke()
```

```
    def message(self):
        print '%s guessed %d, got %d' % \
              (self.name, self.guess, self.throw)

    def broke(self):
        if self.capital == 0:
            print '%s lost!' % self.name
            sys.exit(0)  # end the program
```

The guesses of the computer and the player are specified by functions:

```
def computer_guess(ndice):
    # any of the outcomes (sum) is equally likely:
    return random_number.randint(ndice, 6*ndice)

def player_guess(ndice):
    return input('Guess the sum of the no of eyes '\
                 'in the next throw: ')
```

The key function to play the whole game, utilizing the `Player` class for the computer and the user, can be expressed as

```
def play(nrounds, ndice=2):
    player = Player('YOU', nrounds, player_guess, ndice)
    computer = Player('Computer', nrounds, computer_guess, ndice)

    for i in range(nrounds):
        player.play_one_round()
        computer.play_one_round()
        print 'Status: user have %d euro, machine has %d euro\n' % \
              (player.capital, computer.capital)

    if computer.capital > player.capital:
        winner = 'Machine'
    else:
        winner = 'You'
    print winner, 'won!'
```

The complete code is found in the file `ndice2.py`. There is no new functionality compared to the `ndice.py` implementation, just a new and better structuring of the code.

## 8.5 Monte Carlo Integration

One of the earliest applications of random numbers was numerical computation of integrals, that is, a non-random (deterministic) problem. Here we shall address two related methods for computing $\int_a^b f(x)dx$.

### 8.5.1 Standard Monte Carlo Integration

Let $x_1, \ldots, x_n$ be uniformly distributed random numbers between $a$ and $b$. Then

$$(b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i) \tag{8.7}$$

is an approximation to the integral $\int_a^b f(x)dx$. This method is usually referred to as *Monte Carlo integration*. It is easy to interpret (8.7). A well-known result from calculus is that the integral of a function $f$ over $[a, b]$ equals the mean value of $f$ over $[a, b]$ multiplied by the length of the interval, $b - a$. If we approximate the mean value of $f(x)$ by the mean of $n$ randomly distributed function evaluations $f(x_i)$, we get the method (8.7).

We can implement (8.7) in a small function:

```
import random as random_number

def MCint(f, a, b, n):
    s = 0
    for i in range(n):
        x = random_number.uniform(a, b)
        s += f(x)
    I = (float(b-a)/n)*s
    return I
```

One normally needs a large $n$ to obtain good results with this method, so a faster vectorized version of the `MCint` function is handy:

```
from numpy import *

def MCint_vec(f, a, b, n):
    x = random.uniform(a, b, n)
    s = sum(f(x))
    I = (float(b-a)/n)*s
    return I
```

Let us try the Monte Carlo integration method on a simple linear function $f(x) = 2 + 3x$, integrated from 1 to 2. Most other numerical integration methods will integrate such a linear function exactly, regardless of the number of function evaluations. This is not the case with Monte Carlo integration. It would be interesting to see how the quality of the Monte Carlo approximation increases $n$. To plot the evolution of the integral approximation we must store intermediate `I` values. This requires a slightly modified `MCint` method:

```
def MCint2(f, a, b, n):
    s = 0
    # store the intermediate integral approximations in an
    # array I, where I[k-1] corresponds to k function evals.
    I = zeros(n)
    for k in range(1, n+1):
        x = random_number.uniform(a, b)
        s += f(x)
        I[k-1] = (float(b-a)/k)*s
    return I
```

Note that we let `k` go from 1 to `n` while the indices in `I`, as usual, go from 0 to `n-1`. Since `n` can be very large, the `I` array may consume more memory than what we have on the computer. Therefore, we decide to store only every $N$ values of the approximation. Determining if a value is to be stored or not can then be computed by the mod function (see page 423 or Exercise 2.45):

```
for k in range(1, n+1):
    ...
    if k % N == 0:
        # store
```

That is, every time $k$ can be divided by $N$ without any remainder, we store the value. The complete function takes the following form:

```
def MCint3(f, a, b, n, N=100):
    s = 0
    # store every N intermediate integral approximations in an
    # array I and record the corresponding k value
    I_values = []
    k_values = []
    for k in range(1, n+1):
        x = random_number.uniform(a, b)
        s += f(x)
        if k % N == 0:
            I = (float(b-a)/k)*s
            I_values.append(I)
            k_values.append(k)
    return k_values, I_values
```

Now we have the tools to plot the error in the Monte Carlo approximation as a function of $n$:

```
def f1(x):
    return 2 + 3*x

k, I = MCint3(f1, 1, 2, 1000000, N=10000)
from scitools.std import plot
error = 6.5 - array(I)
plot(k, error, title='Monte Carlo integration',
     xlabel='n', ylabel='error')
```

Figure 8.4 shows the resulting plot.

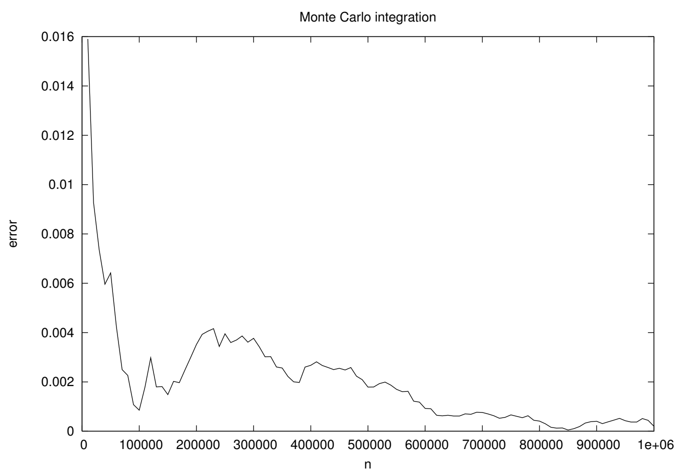



**Fig. 8.4** The convergence of Monte Carlo integration applied to $\int_1^2 (2+3x)dx$.

For functions of one variable, method (8.7) requires many points and is inefficient compared to other integration rules. Most integra-

tion rules have an error that reduces with increasing $n$, typically like $n^{-r}$ for some $r > 0$. For the Trapezoidal rule, $r = 2$, while $r = 1/2$ for Monte Carlo integration, which means that this method converges quite slowly compared to the Trapezoidal rule. However, for functions of many variables, Monte Carlo integration in high space dimension completely outperforms methods like the Trapezoidal rule and Simpson's rule. There are also many ways to improve the performance of (8.7), basically by being "smart" in drawing the random numbers (this is called variance reducing techniques).

### 8.5.2 Computing Areas by Throwing Random Points

Think of some geometric region $G$ in the plane and a surrounding bounding box $B$ with geometry $[x_L, x_H] \times [y_L, y_H]$. One way of computing the area of $G$ is to draw $N$ random points inside $B$ and count how many of them, $M$, that lie inside $G$. The area of $G$ is then the fraction $M/N$ ($G$'s fraction of $B$'s area) times the area of $B$, $(x_H - x_L)(y_H - y_L)$. Phrased differently, this method is a kind of dart game where you record how many hits there are inside $G$ if every throw hits uniformly within $B$.

Let us formulate this method for computing the integral $\int_a^b f(x)dx$. The important observation is that this integral is the area under the curve $y = f(x)$ and above the $x$ axis, between $x = a$ and $x = b$. We introduce a rectangle $B$,

$$B = \{(x, y) \,|\, a \le x \le b,\ 0 \le y \le m\},$$

where $m \le \max_{x\in[a,b]} f(x)$. The algorithm for computing the area under the curve is to draw $N$ random points inside $B$ and count how many of them, $M$, that are above the $x$ axis and below the $y = f(x)$ curve, see Figure 8.5. The area or integral is then estimated by

$$\frac{M}{N} m(b-a)\,.$$

First we implement the "dart method" by a simple loop over points:

```
def MCint_area(f, a, b, n, m):
    below = 0  # counter for no of points below the curve
    for i in range(n):
        x = random_number.uniform(a, b)
        y = random_number.uniform(0, m)
        if y <= f(x):
            below += 1
    area = below/float(n)*m*(b-a)
    return area
```

Note that this method draws twice as many random numbers as the previous method.

A vectorized implementation reads

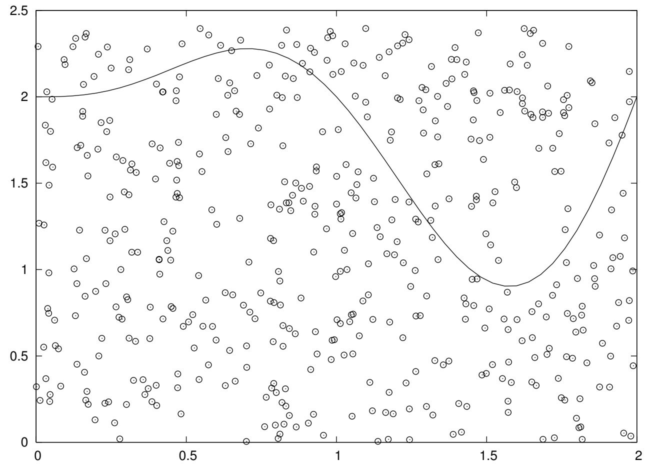



**Fig. 8.5** The "dart" method for computing integrals. When $M$ out of $N$ random points in the rectangle $[0, 2] \times [0, 2.4]$ lie under the curve, the area under the curve is estimated as the $M/N$ fraction of the area of the rectangle, i.e., $(M/N)2 \cdot 2.4$.

```
def MCint_area_vec(f, a, b, n, m):
    x = random.uniform(a, b, n)
    y = random.uniform(0, m, n)
    below = y[y < f(x)].size
    area = below/float(n)*m*(b-a)
    return area
```

Even for 2 million random numbers the plain loop version is not that slow as it executes within some seconds on a slow laptop. Nevertheless, if you need the integration being repeated many times inside another calculation, the superior efficiency of the vectorized version may be important. We can quantify the efficiency gain by the aid of the timer `time.clock()` in the following way (see Appendix E.6.1):

```
import time
t0 = time.clock()
print MCint_area(f1, a, b, n, fmax)
t1 = time.clock()   # time of MCint_area is t1-t0
print MCint_area_vec(f1, a, b, n, fmax)
t2 = time.clock()   # time of MCint_area_vec is t2-t1
print 'loop/vectorized fraction:', (t1-t0)/(t2-t1)
```

With $n = 10^6$ I achieved a factor of about 16 in favor of the vectorized version on an IBM laptop.

## 8.6 Random Walk in One Space Dimension

In this section we shall simulate a collection of particles that move around in a random fashion. This type of simulations are fundamental in physics, biology, chemistry as well as other sciences and can be used to describe many phenomena. Some application areas include molecular

motion, heat conduction, quantum mechanics, polymer chains, population genetics, brain research, hazard games, and pricing of financial instruments.

Imagine that we have some particles that perform random moves, either to the right or to the left. We may flip a coin to decide the movement of each particle, say head implies movement to the right and tail means movement to the left. Each move is one unit length. Physicists use the term *random walk* for this type of movement of a particle[8].

The movement is also known as "drunkard's walk". You may have experienced this after a very wet night on a pub: you step forward and backward in a random fashion. Since these movements on average make you stand still, and since you know that you normally reach home within reasonable time, the model is not good for a real walk. We need to add a *drift* to the walk, so the probability is greater for going forward than backward. This is an easy adjustment, see Exercise 8.33. What may come as a surprise is the following fact: even when there is equal probability of going forward and backward, one can prove mathematically that the drunkard will always reach his home. Or more precisely, he will get home in finite time ("almost surely" as the mathematicians must add to this statement). Exercise 8.34 asks you to experiment with this fact. For many practical purposes, "finite time" does not help much as there might be more steps involved than the time it takes to get sufficiently sober to remove the completely random component of the walk.

### 8.6.1 Basic Implementation

How can we implement $n_s$ random steps of $n_p$ particles in a program? Let us introduce a coordinate system where all movements are along the $x$ axis. An array of $x$ values then holds the positions of all particles. We draw random numbers to simulate flipping a coin, say we draw from the integers 1 and 2, where 1 means head (movement to the right) and 2 means tail (movement to the left). We think the algorithm is conveniently expressed directly as a complete Python program:

```
import random as random_number
import numpy
np = 4                          # no of particles
ns = 100                        # no of steps
positions = numpy.zeros(np)     # all particles start at x=0
HEAD = 1;  TAIL = 2             # constants

for step in range(ns):
    for p in range(np):
        coin = random_number.randint(1,2)  # flip coin
```

[8] You may try this yourself: flip the coin and make one step to the left or right, and repeat this process.

```
        if coin == HEAD:
            positions[p] += 1   # one unit length to the right
        elif coin == TAIL:
            positions[p] -= 1   # one unit length to the left
```

This program is found in the file `walk1D.py`.

### 8.6.2 Visualization

We may add some visualization of the movements by inserting a `plot` command at the end of the `step` loop and a little pause to better separate the frames in the animation[9]:

```
    plot(positions, y, 'ko3', axis=[xmin, xmax, -0.2, 0.2])
    time.sleep(0.2)  # pause
```

Recall from Chapter 4 that in an animation like this the axis must be kept fixed. We know that in $n_s$ steps, no particle can move longer than $n_s$ unit lengths to the right or to the left so the extent of the $x$ axis becomes $[-n_s, n_s]$. However, the probability of reaching these lower or upper limit is very small[10]. Most of the movements will take place in the center of the plot. We may therefore shrink the extent of the axis to better view the movements. It is known that the expected extent of the particles is of the order $\sqrt{n_s}$, so we may take the maximum and minimum values in the plot as $\pm 2\sqrt{n_s}$. However, if a position of a particle exceeds these values, we extend `xmax` and `xmin` by $2\sqrt{n_s}$ in positive and negative $x$ direction, respectively.

The $y$ positions of the particles are taken as zero, but it is necessary to have some extent of the $y$ axis, otherwise the coordinate system collapses and most plotting packages will refuse to draw the plot. Here we have just chosen the $y$ axis to go from -0.2 to 0.2. You can find the complete program in `src/random/walk1Dp.py`. The `np` and `ns` parameters can be set as the first two command-line arguments:

Terminal

```
walk1Dp.py 6 200
```

It is hard to claim that this program has astonishing graphics. In Chapter 8.7, where we let the particles move in two space dimensions, the graphics gets much more exciting.

### 8.6.3 Random Walk as a Difference Equation

The random walk process can easily be expressed in terms of a difference equation (Chapter 5). Let $x_n$ be the position of the particle at

[9] These actions require `from scitools.std import *` and `import time`.

[10] The probability is $2^{-n_s}$, which becomes about $10^{-9}$ for 30 steps.

time $n$. This position is an evolvement from time $n-1$, obtained by adding a random variable $s$ to the previous position $x_{n-1}$, where $s=1$ has probability 1/2 and $s=-1$ has probability 1/2. In statistics, the expression "probability of event A" is written $\mathrm{P}(A)$. We can therefore write $\mathrm{P}(s=1)=1/2$ and $\mathrm{P}(s=-1)=1/2$. The difference equation can now be expressed mathematically as

$$x_n = x_{n-1} + s, \quad x_0 = 0, \quad \mathrm{P}(s=1) = \mathrm{P}(s=-1) = 1/2\,. \qquad (8.8)$$

This equation governs the motion of one particle. For a collection $m$ of particles we introduce $x_n^{(i)}$ as the position of the $i$-th particle at the $n$-th time step. Each $x_n^{(i)}$ is governed by (8.8), and all the $s$ values in each of the $m$ difference equations are independent of each other.

### 8.6.4 Computing Statistics of the Particle Positions

Scientists interested in random walks are in general not interested in the graphics of our `walk1D.py` program, but more in the statistics of the positions of the particles at each step. We may therefore, at each step, compute a histogram of the distribution of the particles along the $x$ axis, plus estimate the mean position and the standard deviation. These mathematical operations are easily accomplished by letting the SciTools function `compute_histogram` and the `numpy` functions `mean` and `std` operate on the `positions` array (see Chapter 8.1.5)[11] :

```
mean_pos  = mean(positions)
stdev_pos = std(positions)
pos, freq = compute_histogram(positions, nbins=int(xmax),
                              piecewise_constant=True)
```

We can plot the particles as circles, as before, and add the histogram and vertical lines for the mean and the positive and negative standard deviation (the latter indicates the "width" of the distribution of particles). The vertical lines can be defined by the six lists

```
xmean, ymean   = [mean_pos, mean_pos],     [yminv, ymaxv]
xstdv1, ystdv1 = [stdev_pos, stdev_pos],   [yminv, ymaxv]
xstdv2, ystdv2 = [-stdev_pos, -stdev_pos], [yminv, ymaxv]
```

where `yminv` and `ymaxv` are the minimum and maximum $y$ values of the vertical lines. The following command plots the position of every particle as circles, the histogram as a curve, and the vertical lines with a thicker line:

[11] The number of bins in the histogram is just based on the extent of the particles. It could also have been a fixed number.

```
    plot(positions, y, 'ko3',      # particles as circles
         pos, freq, 'r',           # histogram
         xmean, ymean, 'r2',       # mean position as thick line
         xstdv1, ystdv1, 'b2',     # +1 standard dev.
         xstdv2, ystdv2, 'b2',     # -1 standard dev.
         axis=[xmin, xmax, ymin, ymax],
         title='random walk of %d particles after %d steps' % \
               (np, step+1))
```

This plot is then created at every step in the random walk. By observing the graphics, one will soon realize that the computation of the extent of the $y$ axis in the plot needs some considerations. We have found it convenient to base `ymax` on the maximum value of the histogram (`max(freq)`), plus some space (chosen as 10 percent of `max(freq)`). However, we do not change the `ymax` value unless it is more than 0.1 different from the previous `ymax` value (otherwise the axis "jumps" too often). The minimum value, `ymin`, is set to `ymin=-0.1*ymax` every time we change the `ymax` value. The complete code is found in the file `walk1Ds.py`. If you try out 2000 particles and 30 steps, the final graphics becomes like that in Figure 8.6. As the number of steps is increased, the particles are dispersed in the positive and negative $x$ direction, and the histogram gets flatter and flatter. Letting $\hat{H}(i)$ be the histogram value in interval number $i$, and each interval having width $\Delta x$, the probability of finding a particle in interval $i$ is $\hat{H}(i)\Delta x$. It can be shown mathematically that the histogram is an approximation to the probability density function of the normal distribution (1.6) (see page 45), with mean zero and standard deviation $s \sim \sqrt{n}$, where $n$ is the step number.

### 8.6.5 Vectorized Implementation

There is no problem with the speed of our one-dimensional random walkers in the `walk1Dp.py` or `walk1Ds.py` programs, but in real-life applications of such simulation models, we often have a very large number of particles performing a very large number of steps. It is then important to make the implementation as efficient as possible. Two loops over all particles and all steps, as we have in the programs above, become very slow compared to a vectorized implementation.

A vectorized implementation of a one-dimensional walk should utilize the functions `randint` or `random_integers` from `numpy.random`. A first idea may be to draw steps for all particles at a step simultaneously. Then we repeat this process in a loop from 0 to $n_s - 1$. However, these repetitions are just new vectors of random numbers, and we may avoid the loop if we draw $n_p \times n_s$ random numbers at once:

```
moves = random.randint(1, 3, size=np*ns)
# or
moves = random.random_integers(1, 2, size=np*ns)
```

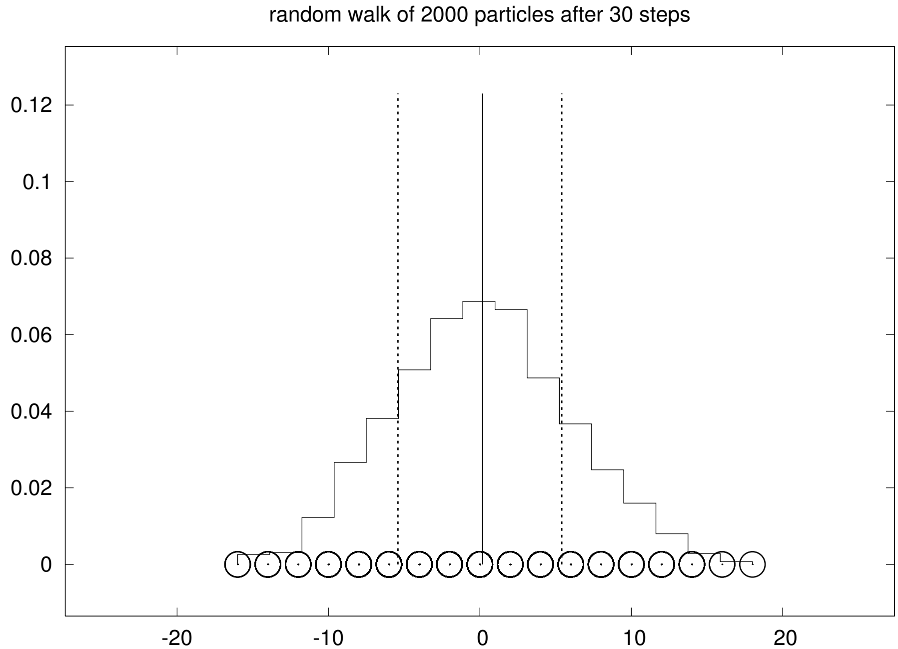



**Fig. 8.6** Particle positions (circles), histogram (piecewise constant curve), and vertical lines indicating the mean value and the standard deviation from the mean after a one-dimensional random walk of 2000 particles for 30 steps.

The values are now either 1 or 2, but we want −1 or 1. A simple scaling and translation of the numbers transform the 1 and 2 values to −1 and 1 values:

```
moves = 2*moves - 3
```

Then we can create a two-dimensional array out of `moves` such that `moves[i,j]` is the `i`-th step of particle number `j`:

```
moves.shape = (ns, np)
```

It does not make sense to plot the evolution of the particles and the histogram in the vectorized version of the code, because the point with vectorization is to speed up the calculations, and the visualization takes much more time than drawing random numbers, even in the `walk1Dp.py` and `walk1Ds.py` programs from Chapter 8.6.4. We therefore just compute the positions of the particles inside a loop over the steps and some simple statistics. At the end, after $n_s$ steps, we plot the histogram of the particle distribution along with circles for the positions of the particles. The rest of the program, found in the file `walk1Dv.py`, looks as follows:

```
positions = zeros(np)
for step in range(ns):
    positions += moves[step, :]

    mean_pos, stdev_pos = mean(positions), std(positions)
    print mean_pos, stdev_pos

nbins = int(3*sqrt(ns))    # no of intervals in histogram
pos, freq = compute_histogram(positions, nbins,
                              piecewise_constant=True)

plot(positions, zeros(np), 'ko3',
     pos, freq, 'r',
     axis=[min(positions), max(positions), -0.05, 1.1*max(freq)],
     hardcopy='tmp.ps')
```

## 8.7 Random Walk in Two Space Dimensions

A random walk in two dimensions performs a step either to the north, south, west, or east, each one with probability 1/4. To demonstrate this process, we introduce $x$ and $y$ coordinates of $n_p$ particles and draw random numbers among 1, 2, 3, or 4 to determine the move. The positions of the particles can easily be visualized as small circles in an $xy$ coordinate system.

### 8.7.1 Basic Implementation

The algorithm described above is conveniently expressed directly as a complete working program:

```
def random_walk_2D(np, ns, plot_step):
    xpositions = zeros(np)
    ypositions = zeros(np)
    # extent of the axis in the plot:
    xymax = 3*sqrt(ns); xymin = -xymax

    NORTH = 1;  SOUTH = 2;  WEST = 3;  EAST = 4  # constants

    for step in range(ns):
        for i in range(np):
            direction = random_number.randint(1, 4)
            if direction == NORTH:
                ypositions[i] += 1
            elif direction == SOUTH:
                ypositions[i] -= 1
            elif direction == EAST:
                xpositions[i] += 1
            elif direction == WEST:
                xpositions[i] -= 1

        # plot just every plot_step steps:
        if (step+1) % plot_step == 0:
            plot(xpositions, ypositions, 'ko',
                 axis=[xymin, xymax, xymin, xymax],
                 title='%d particles after %d steps' % \
```

```
                        (np, step+1),
                 hardcopy='tmp_%03d.eps' % (step+1))
    return xpositions, ypositions

# main program:
import random as random_number
random_number.seed(10)
import sys
from scitools.std import zeros, plot, sqrt

np        = int(sys.argv[1])  # number of particles
ns        = int(sys.argv[2])  # number of steps
plot_step = int(sys.argv[3])  # plot every plot_step steps
x, y = random_walk_2D(np, ns, plot_step)
```

The program is found in the file `walk2D.py`. Figure 8.7 shows two snapshots of the distribution of 3000 particles after 40 and 400 steps. These plots were generated with command-line arguments `3000 400 20`, the latter implying that we visualize the particles every 20 time steps only.

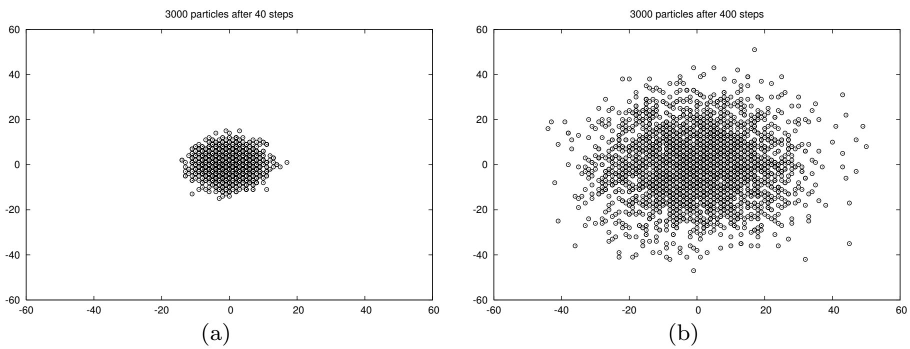



**Fig. 8.7** Location of 3000 particles starting at the origin and performing a random walk: (a) 40 steps; (b) 400 steps.

To get a feeling for the two-dimensional random walk you can try out only 30 particles for 400 steps and let each step be visualized (i.e., command-line arguments `30 400 1`). The update of the movements is now fast.

The `walk2D.py` program dumps the plots to PostScript files with names of the form `tmp_xxx.eps`, where `xxx` is the step number. We can create a movie out of these individual files using the `movie` function (Chapter 4.3.7) or the program `convert` from the ImageMagick suite[12]:

```
convert -delay 50 -loop 1000 tmp_*.eps movie.gif
```

All the plots are now put after each other as frames in a movie, with a delay of 50 ms between each frame. The movie will run in a loop 1000 times. Alternatively, we can create the movie with the `movie` function from Easyviz, inside a program:

[12] If you want to run this command from an IPython session, prefix `convert` with an exclamation mark: `!convert`.

```
from scitools.std import movie
movie('tmp_*.eps', encoder='convert', output_file='movie.gif')
```

The resulting movie file is named `movie.gif`, which can be viewed by the `animate` program (also from the ImageMagick program suite), just write `animate movie.gif`. Making and showing the movie are slow processes if a large number of steps are included in the movie – 100 steps or fewer are appropriate, but this depends on the power of your computer.

### 8.7.2 Vectorized Implementation

The `walk2D.py` program is quite slow. Now the visualization is much faster than the movement of the particles. Vectorization may speed up the `walk2D.py` program significantly. As in the one-dimensional phase, we draw all the movements at once and then invoke a loop over the steps to update the $x$ and $y$ coordinates. We draw $n_s \times n_p$ numbers among 1, 2, 3, and 4. We then reshape the vector of random numbers to a two-dimensional array `moves[i, j]`, where `i` counts the steps, `j` counts the particles. The `if` test on whether the current move is to the north, south, east, or west can be vectorized using the `where` function (see Chapter 4.4.1). For example, if the random numbers for all particles in the current step are accessible in an array `this_move`, we could update the $x$ positions by

```
xpositions += where(this_move == EAST, 1, 0)
xpositions -= where(this_move == WEST, 1, 0)
```

provided `EAST` and `WEST` are constants, equal to 3 and 4, respectively. A similar construction can be used for the $y$ moves.

The complete program is listed below:

```
def random_walk_2D(np, ns, plot_step):
    xpositions = zeros(np)
    ypositions = zeros(np)
    moves = random.random_integers(1, 4, size=ns*np)
    moves.shape = (ns, np)

    # estimate max and min positions:
    xymax = 3*sqrt(ns); xymin = -xymax

    NORTH = 1;  SOUTH = 2;  WEST = 3;  EAST = 4  # constants

    for step in range(ns):
        this_move = moves[step,:]
        ypositions += where(this_move == NORTH, 1, 0)
        ypositions -= where(this_move == SOUTH, 1, 0)
        xpositions += where(this_move == EAST,  1, 0)
        xpositions -= where(this_move == WEST,  1, 0)

        # just plot every plot_step steps:
        if (step+1) % plot_step == 0:
            plot(xpositions, ypositions, 'ko',
                 axis=[xymin, xymax, xymin, xymax],
```

```
                 title='%d particles after %d steps' % \
                       (np, step+1),
                 hardcopy='tmp_%03d.eps' % (step+1))
    return xpositions, ypositions

# main program:
from scitools.std import *
random.seed(11)

np = int(sys.argv[1])  # number of particles
ns = int(sys.argv[2])  # number of steps
plot_step = int(sys.argv[3])  # plot each plot_step step
x, y = random_walk_2D(np, ns, plot_step)
```

You will easily experience that this program, found in the file `walk2Dv.py`, runs significantly faster than the `walk2D.py` program.

## 8.8 Summary

### 8.8.1 Chapter Topics

*Drawing Random Numbers.* Random numbers can be scattered throughout an interval in various ways, specified by the *distribution* of the numbers. We have considered a uniform distribution (Chapter 8.1.2) and a normal (or Gaussian) distribution (Chapter 8.1.6). Table 8.1 shows the syntax for generating random numbers of these two distributions, using either the standard scalar `random` module in Python or the vectorized `numpy.random` module.

**Table 8.1** Summary of important functions for drawing random numbers. `N` is the array length in vectorized drawing, while `m` and `s` represent the mean and standard deviation values of a normal distribution.

| | `random` | `numpy.random` |
|---|---|---|
| uniform numbers in $[0, 1)$ | `random()` | `random(N)` |
| uniform numbers in $[a, b)$ | `uniform(a, b)` | `uniform(a, b, N)` |
| integers in $[a, b]$ | `randint(a, b)` | `randint(a, b+1, N)` |
| | | `random_integers(a, b, N)` |
| Gaussian numbers, mean $m$, st.dev. $s$ | `gauss(m, s)` | `normal(m, s, N)` |
| set seed (`i`) | `seed(i)` | `seed(i)` |
| shuffle list `a` (in-place) | `shuffle(a)` | `shuffle(a)` |
| choose a random element in list `a` | `choice(a)` | |

*Typical Probability Computation.* Many programs performing probability computations draw a large number `N` of random numbers and count how many times `M` a random number leads to some true condition (Monte Carlo simulation):

```
import random as random_number
M = 0
for i in xrange(N):
    r = random_number.randint(a, b)
    if condition:
        M += 1
print 'Probability estimate:', float(M)/N
```

For example, if we seek the probability that we get at least four eyes when throwing a dice, we choose the random number to be the number of eyes, i.e., an integer in the interval $[1, 6]$ (`a=1`, `b=6`) and `condition` becomes `r >= 4`.

For large `N` we can speed up such programs by vectorization, i.e., drawing all random numbers at once in a big array and use operations on the array to find $M$. The similar vectorized version of the program above looks like

```
from numpy import *
r = random.uniform(a, b, N)
M = sum(condition)
# or
M = sum(where(condition, 1, 0))
print 'Probability estimate:', float(M)/N
```

(Combinations of boolean expressions in the `condition` argument to `where` requires special constructs as outlined in Exercise 8.14.) Make sure you use `sum` from `numpy`, when operating on large arrays, and not the much slower built-in `sum` function in Python.

*Statistical Measures.* Given an array of random numbers, the following code computes the mean, variance, and standard deviation of the numbers and finally displays a plot of the histogram, which reflects how the numbers are statistically distributed:

```
from scitools.std import mean, var, std, compute_histogram
m = mean(numbers)
v = var(numbers)
s = std(numbers)
x, y = compute_histogram(numbers, 50, piecewise_constant=True)
plot(x, y)
```

### 8.8.2 Summarizing Example: Random Growth

Chapter 5.1.1 contains mathematical models for how an investment grows when there is an interest rate being added to the investment at certain intervals. The model can easily allow for a time-varying interest rate, but for forecasting the growth of an investment, it is difficult to predict the future interest rate. One commonly used method is to build a probabilistic model for the development of the interest rate, where the rate is chosen randomly at random times. This gives a random

growth of the investment, but by simulating many random scenarios we can compute the mean growth and use the standard deviation as a measure of the uncertainty of the predictions.

*Problem.* Let $p$ be the annual interest rate in a bank in percent. Suppose the interest is added to the investment $q$ times per year. The new value of the investment, $x_n$, is given by the previous value of the investment, $x_{n-1}$, plus the $p/q$ percent interest:

$$x_n = x_{n-1} + \frac{p}{100q} x_{n-1} \,.$$

Normally, the interest is added daily ($q = 360$ and $n$ counts days), but for efficiency in the computations later we shall assume that the interest is added monthly, so $q = 12$ and $n$ counts months.

The basic assumption now is that $p$ is random and varies with time. Suppose $p$ increases with a random amount $\gamma$ from one month to the next:

$$p_n = p_{n-1} + \gamma \,.$$

A typical size of $p$ adjustments is 0.5. However, the central bank does not adjust the interest every month. Instead this happens every $M$ months on average. The probability of a $\gamma \neq 0$ can therefore be taken as $1/M$. In a month where $\gamma \neq 0$, we may say that $\gamma = m$ with probability 1/2 or $\gamma = -m$ with probability 1/2 if it is equally likely that the rate goes up as down (this is not a good assumption, but a more complicated evolvement of $\gamma$ is postponed now).

*Solution.* First we must develop the precise formulas to be implemented. The difference equations for $x_n$ and $p_n$ are in simple in the present case, but the details computing $\gamma$ must be worked out. In a program, we can draw two random numbers to estimate $\gamma$: one for deciding if $\gamma \neq 0$ and the other for determining the sign of the change. Since the probability for $\gamma \neq 0$ is $1/M$, we can draw a number $r_1$ among the integers $1, \ldots, M$ and if $r_1 = 1$ we continue with drawing a second number $r_2$ among the integers 1 and 2. If $r_2 = 1$ we set $\gamma = m$, and if $r_2 = 2$ we set $\gamma = -m$. We must also assure that $p_n$ does not take on unreasonable values, so we choose $p_n < 1$ and $p_n > 15$ as cases where $p_n$ is not changed.

The mathematical model for the investment must track both $x_n$ and $p_n$. Below we express with precise mathematics the equations for $x_n$ and $p_n$ and the computation of the random $\gamma$ quantity:

$$
\begin{aligned}
x_n &= x_{n-1} + \frac{p_{n-1}}{12 \cdot 100} x_{n-1}, \quad i = 1, \ldots, N && (8.9)\\
r_1 &= \text{random integer in } [1, M] && (8.10)\\
r_2 &= \text{random integer in } [1, 2] && (8.11)\\
\gamma &= \begin{cases} m, & \text{if } r_1 = 1 \text{ and } r_2 = 1, \\ -m, & \text{if } r_1 = 1 \text{ and } r_2 = 2, \\ 0, & \text{if } r_1 \neq 1 \end{cases} && (8.12)\\
p_n &= p_{n-1} + \begin{cases} \gamma, & \text{if } p_{n-1} + \gamma \in [1, 15], \\ 0, & \text{otherwise} \end{cases} && (8.13)
\end{aligned}
$$

We remark that the evolution of $p_n$ is much like a random walk process (Chapter 8.6), the only differences is that the plus/minus steps are taken at some random points among the times $0, 1, 2, \ldots, N$ rather than at all times $0, 1, 2, \ldots, N$. The random walk for $p_n$ also has barriers at $p = 1$ and $p = 15$, but that is common in a standard random walk too.

Each time we calculate the $x_n$ sequence in the present application, we get a different development because of the random numbers involved. We say that one development of $x_0, \ldots, x_n$ is a *path* (or realization, but since the realization can be viewed as a curve $x_n$ or $p_n$ versus $n$ in this case, it is common to use the word path). Our Monte Carlo simulation approach consists of computing a large number of paths, as well as the sum of the path and the sum of the paths squared. From the latter two sums we can compute the mean and standard deviation of the paths to see the average development of the investment and the uncertainty of this development. Since we are interested in complete paths, we need to store the complete sequence of $x_n$ for each path. We may also be interested in the statistics of the interest rate so we store the complete sequence $p_n$ too.

Programs should be built in pieces so that we can test each piece before testing the whole program. In the present case, a natural piece is a function that computes one path of $x_n$ and $p_n$ with $N$ steps, given $M$, $m$, and the initial conditions $x_0$ and $p_0$. We can then test this function before moving on to calling the function a large number of times. An appropriate code may be

```
def simulate_one_path(N, x0, p0, M, m):
    x = zeros(N+1)
    p = zeros(N+1)
    index_set = range(0, N+1)

    x[0] = x0
    p[0] = p0

    for n in index_set[1:]:
        x[n] = x[n-1] + p[n-1]/(100.0*12)*x[n-1]

        # update interest rate p:
        r = random_number.randint(1, M)
        if r == 1:
```

```
            # adjust gamma:
            r = random_number.randint(1, 2)
            gamma = m if r == 1 else -m
        else:
            gamma = 0
        pn = p[n-1] + gamma
        p[n] = pn if 1 <= pn <= 15 else p[n-1]
    return x, p
```

Testing such a function is challenging because the result is different each time because of the random numbers. A first step in verifying the implementation is to turn off the randomness ($m = 0$) and check that the deterministic parts of the difference equations are correctly computed:

```
x, p = simulate_one_path(3, 1, 10, 1, 0)
print x
```

The output becomes

```
[ 1.          1.00833333  1.01673611  1.02520891]
```

These numbers can quickly be checked against a formula of the type (5.4) on page 237 in an interactive session:

```
>>> def g(x0, n, p):
...     return x0*(1 + p/(12.*100))**n
...
>>> g(1, 1, 10)
1.0083333333333333
>>> g(1, 2, 10)
1.0167361111111111
>>> g(1, 3, 10)
1.0252089120370369
```

We can conclude that our function works well when there is no randomness. A next step is to carefully examine the code that computes `gamma` and compare with the mathematical formulas.

Simulating many paths and computing the average development of $x_n$ and $p_n$ is a matter of calling `simulate_one_path` repeatedly, use two arrays `xm` and `pm` to collect the sum of `x` and `p`, respectively, and finally obtain the average path by dividing `xm` and `pm` by the number of paths we have computed:

```
def simulate_n_paths(n, N, L, p0, M, m):
    xm = zeros(N+1)
    pm = zeros(N+1)
    for i in range(n):
        x, p = simulate_one_path(N, L, p0, M, m)
        # accumulate paths:
        xm += x
        pm += p
    # compute average:
    xm /= float(n)
    pm /= float(n)
    return xm, pm
```

We can also compute the standard deviation of the paths using formulas (8.3) and (8.6), with $x_j$ as either an `x` or a `p` array. It might

happen that small round-off errors generate a small *negative* variance, which mathematically should have been slightly greater than zero. Taking the square root will then generate complex arrays and problems with plotting. To avoid this problem, we therefore replace all negative elements by zeros in the variance arrays before taking the square root. The new lines for computing the standard deviation arrays `xs` and `ps` are indicated below:

```
def simulate_n_paths(n, N, x0, p0, M, m):
    ...
    xs = zeros(N+1)  # standard deviation of x
    ps = zeros(N+1)  # standard deviation of p
    for i in range(n):
        x, p = simulate_one_path(N, x0, p0, M, m)
        # accumulate paths:
        xm += x
        pm += p
        xs += x**2
        ps += p**2

    ...
    # compute standard deviation:
    xs = xs/float(n) - xm*xm  # variance
    ps = ps/float(n) - pm*pm  # variance
    # remove small negative numbers (round off errors):
    xs[xs < 0] = 0
    ps[ps < 0] = 0
    xs = sqrt(xs)
    ps = sqrt(ps)
    return xm, xs, pm, ps
```

A remark regarding the efficiency of array operations is appropriate here. The statement `xs += x**2` could equally well, from a mathematical point of view, be written as `xs = xs + x**2`. However, in this latter statement, two extra arrays are created (one for the squaring and one for the sum), while in the former only one array (`x**2`) is made. Since the paths can be long and we make many simulations, such optimizations can be important.

One may wonder whether `x**2` is "smart" in the sense that squaring is detected and computed as `x*x`, not as a general (slow) power function. This is indeed the case for arrays, as we have investigated in the little test program `smart_power.py` in the `random` directory. This program applies time measurement methods from Appendix E.6.2.

Our `simulate_n_paths` function generates four arrays which are natural to visualize. Having a mean and a standard deviation curve, it is often common to plot the mean curve with one color or linetype and then two curves, corresponding to plus one and minus one standard deviation, with another less visible color. This gives an indication of the mean development and the uncertainty of the underlying process. We therefore make two plots: one with `xm`, `xm+xs`, and `xm-xs`, and one with `pm`, `pm+ps`, and `pm-ps`.

Both for debugging and curiosity it is handy to have some plots of a few actual paths. We may pick out 5 paths from the simulations and visualize these:

```
def simulate_n_paths(n, N, x0, p0, M, m):
    ...
    for i in range(n):
        ...
        # show 5 random sample paths:
        if i % (n/5) == 0:
            figure(1)
            plot(x, title='sample paths of investment')
            hold('on')
            figure(2)
            plot(p, title='sample paths of interest rate')
            hold('on')
    figure(1); hardcopy('tmp_sample_paths_investment.eps')
    figure(2); hardcopy('tmp_sample_paths_interestrate.eps')
    ...
    return ...
```

Note the use of `figure`: we need to hold on both figures to add new plots and switch between the figures, both for plotting and making the final hardcopy.

After the visualization of sample paths we make the mean ± standard deviation plots by this code:

```
xm, xs, pm, ps = simulate_n_paths(n, N, x0, p0, M, m)
figure(3)
months = range(len(xm))  # indices along the x axis
plot(months, xm, 'r',
     months, xm-xs, 'y',
     months, xm+xs, 'y',
     title='Mean +/- 1 st.dev. of investment',
     hardcopy='tmp_mean_investment.eps')
figure(4)
plot(months, pm, 'r',
     months, pm-ps, 'y',
     months, pm+ps, 'y',
     title='Mean +/- 1 st.dev. of annual interest rate',
     hardcopy='tmp_mean_interestrate.eps')
```

The complete program for simulating the investment development is found in the file `growth_random.py`.

Running the program with the input data

```
x0 = 1                  # initial investment
p0 = 5                  # initial interest rate
N = 10*12               # number of months
M = 3                   # p changes (on average) every M months
n = 1000                # number of simulations
m = 0.5                 # adjustment of p
```

and initializing the seed of the random generator to 1, we get four plots, which are shown in Figure 8.8.

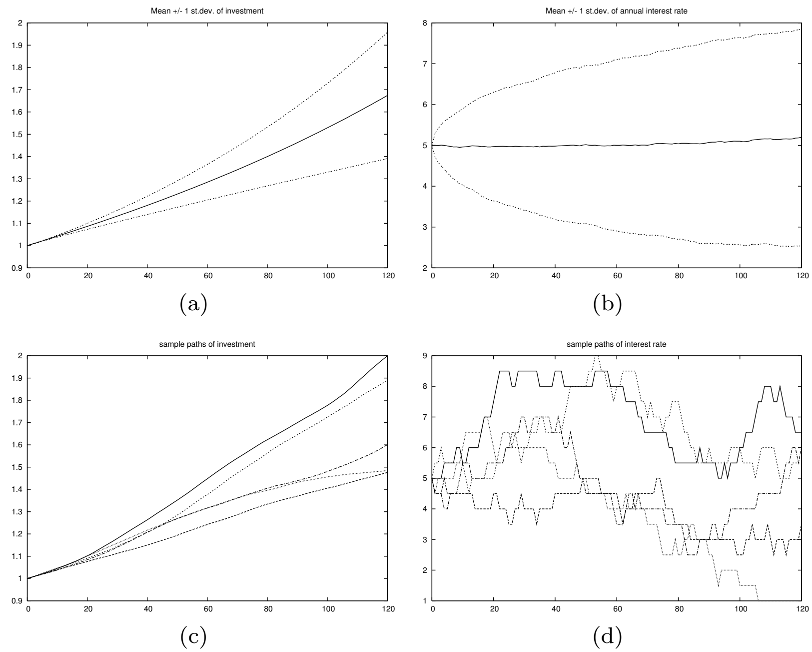



**Fig. 8.8** Development of an investment with random jumps of the interest rate at random points of time: (a) mean value of investment ± one standard deviation; (b) mean value of the interest rate ± one standard deviation; (c) five paths of the investment development; (d) five paths of the interest rate development.

## 8.9 Exercises

**Exercise 8.1.** *Flip a coin $N$ times.*

Make a program that simulates flipping a coin $N$ times. Print out "tail" or "head" for each flip and let the program count the number of heads. (Hint: Use `r = random.random()` and define head as `r <= 0.5` or draw an integer among $\{1, 2\}$ with `r = random.randint(1,2)` and define head when `r` is 1.) Name of program file: `flip_coin.py`. ◇

**Exercise 8.2.** *Compute a probability.*

What is the probability of getting a number between 0.5 and 0.6 when drawing uniformly distributed random numbers from the interval $[0, 1)$? To answer this question empirically, let a program draw $N$ such random numbers using Python's standard `random` module, count how many of them, $M$, that fall in the interval $(0.5, 0.6)$, and compute the probability as $M/N$. Run the program with the four values $N = 10^i$ for $i = 1, 2, 3, 6$. Name of program file: `compute_prob.py`. ◇

**Exercise 8.3.** *Choose random colors.*

Suppose we have eight different colors. Make a program that chooses one of these colors at random and writes out the color. Hint: Use a list of color names and use the `choice` function in the `random` module to pick a list element. Name of program file: `choose_color.py`. ◇

**Exercise 8.4.** *Draw balls from a hat.*

Suppose there are 40 balls in a hat, of which 10 are red, 10 are blue, 10 are yellow, and 10 are purple. What is the probability of getting two blue and two purple balls when drawing 10 balls at random from the hat? Name of program file: `4balls_from10.py`. ◇

**Exercise 8.5.** *Probabilities of rolling dice.*

1. You throw a die. What is the probability of getting a 6?
2. You throw a die four times in a row. What is the probability of getting 6 all the times?
3. Suppose you have thrown the die three times with 6 coming up all times. What is the probability of getting a 6 in the fourth throw?
4. Suppose you have thrown the die 100 times and experienced a 6 in every throw. What do you think about the probability of getting a 6 in the next throw?

First try to solve the questions from a theoretical or common sense point of view. Thereafter, make functions for simulating cases 1, 2, and 3. Name of program file: `rolling_dice.py`. ◇

**Exercise 8.6.** *Estimate the probability in a dice game.*

Make a program for estimating the probability of getting at least one 6 when throwing $n$ dice. Read $n$ and the number of experiments from the command line. (To verify the program, you can compare the estimated probability with the exact result $11/36$ when $n = 2$.) Name of program file: `one6_2dice.py`. ◇

**Exercise 8.7.** *Decide if a dice game is fair.*

Somebody suggests the following game. You pay 1 unit of money and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you win 10 units of money, otherwise you lose your investment. Should you play this game? Answer the question by making a program that simulates the game. Name of program file: `sum9_4dice.py`. ◇

**Exercise 8.8.** *Adjust the game in Exer. 8.7.*

It turns out that the game in Exercise 8.7 is not fair, since you lose money in the long run. The purpose of this exercise is to adjust the winning award so that the game becomes fair, i.e., that you neither lose nor win money in the long run.

Make a program that computes the probability $p$ of getting a sum less than $s$ when rolling $n$ dice. Name of program file: `sum_s_ndice_fair.py`.

If the cost of each game is $q$ units of money, the game is fair if the payment in case you win is $r = q/p$. Run the program you made for $s = 9$ and $n = 4$, which corresponds to the game in Exercise 8.7, and compute the corresponding $p$. Modify the program from Exercise 8.7 so that the award is $r = 1/p$, and run that program to see that now

the game is fair, i.e., you neither win nor lose money in a large number of games.

*Explanation.* The formula for a fair game can be developed as follows. Let $p = M/N$ be the probability of winning, which means that you in the long run win $M$ out of $N$ games. The cost is $Nq$ and the income is $Mr$. To make the net income $Mr - Nq$ zero, which is the requirement of a fair game, we get $r = qN/M = q/p$. (This reasoning is based on common sense and an intuitive interpretation of probability. More precise reasoning from probability theory will introduce the game as an experiment with two outcomes, either you win with probability $p$ and or lose with probability $1-p$. The expexted payment is then the sum of probabilities times the corresponding net incomes: $-q(1-p)+(r-q)p$ (recall that the net income in a winning game is $r - q$). A fair game has zero expected payment, i.e., $r = q/p$.) ◇

**Exercise 8.9.** *Probabilities of throwing two dice.*

Make a computer program for throwing two dice a large number of times. Record the sum of the eyes each time and count how many times each of the possibilities for the sum (2, 3, ..., 12) appear. A dictionary with the sum as key and count as value is convenient here. Divide the counts by the total number of trials such that you get the frequency of each possible sum. Write out the frequencies and compare them with exact probabilities. (To find the exact probabilities, set up all the $6\times 6$ possible outcomes of throwing two dice, and then count how many of them that has a sum $s$ for $s = 2, 3, \ldots, 12$.) Name of program file: `freq_2dice.py`. ◇

**Exercise 8.10.** *Compute the probability of drawing balls.*

A hat has 20 balls, 5 red, 5 yellow, 5 green, and 5 brown. We draw $n \geq 3$ balls at random. What is the probability of getting

- at least one red and one brown ball?
- exactly one red ball?
- exactly two red balls?
- at least three green balls?

Use Monte Carlo simulation to compute the probabilities and write out the answers to the four questions for $n = 3, 5, 7, 10, 15$. Name of program file: `draw_balls.py`. ◇

**Exercise 8.11.** *Compute the probability of hands of cards.*

Use the `Deck.py` module (in `src/random`) and the `same_rank` and `same_suit` functions from the `cards` module to compute the following probabilities by Monte Carlo simulation:

- exactly two pairs among five cards,
- four or five cards of the same suit among five cards,
- four-of-a-kind among five cards.

Name of program file: `card_hands.py`. ◇

**Exercise 8.12.** *Play with vectorized boolean expressions.*

Using the `numpy.random` module, make an array containing $N$ uniformly distributed random numbers between 0 and 1. Print out the arrays `r <= 0.5`, `r[r <= 0.5]`, `where(r <= 0.5, 1, 0)` and convince yourself that you understand what these arrays express. We want to compute how many of the elements in `r` that are less than or equal to 0.5. How can this be done in a vectorized way, i.e., without explicit loops in the program, but solely with operations on complete arrays? Name of program file: `bool_vec.py`. ◇

**Exercise 8.13.** *Vectorize the program from Exer. 8.1.*

Simulate flipping a coin $N$ times and write out the number of tails. The code should be vectorized, i.e., there must be no loops in Python. Hint: Use ideas from Exercise 8.12. Name of program file: `flip_coin_vec.py`. ◇

**Exercise 8.14.** *Vectorize the code in Exer. 8.2.*

The purpose of this exercise is to speed up the code in Exercise 8.2 by vectorization. Hint: First draw an array `r` with a large number of random numbers in $[0, 1)$. The simplest way to count how many elements in `r` that lie between 0.5 and 0.6, is to first extract the elements larger than 0.5: `r1 = r[r>0.5]`, and then extract the elements in `r1` that are less than 0.6 and get the size of this array: `r1[r1<=0.6].size`. Name of program file: `compute_prob_vec.py`.

*Remark.* An alternative and more complicated method is to use the `where` function. The condition (the first argument to `where`) is now a compond boolean expression `0.5 <= r <= 0.6`, but this cannot be used with NumPy arrays. Instead one must test for `0.5 <= r` *and* `r < = 0.6`. The needed boolean construction in the `where` call is `operator.and_(0.5 <= r, r <= 0.6)`. See also the discussion of the same topic in Chapter 4.4.1. ◇

**Exercise 8.15.** *Throw dice and compute a small probability.*

Compute the probability of getting 6 eyes on all dice when rolling 7 dice. Since you need a large number of experiments in this case (see the first paragraph of Chapter 8.3), you can save quite some simulation time by using a vectorized implementation. Name of program file: `roll_7dice.py`. ◇

**Exercise 8.16.** *Difference equation for random numbers.*

Simple random number generators are based on simulating difference equations. Here is a typical set of two equations:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{8.14}$$

$$y_n = x_n/m, \tag{8.15}$$

for $n = 1, 2, \ldots$. A seed $x_0$ must be given to start the sequence. The numbers $y_1, y_2, \ldots,$ represent the random numbers and $x_0, x_1, \ldots$ are "help" numbers. Although $y_n$ is completely deterministic from (8.14)–(8.15), the sequence $y_n$ *appears* random. The mathematical expression $p \bmod q$ is coded as `p % q` in Python.

Use $a = 8121$, $c = 28411$, and $m = 134456$. Solve the system (8.14)–(8.15) in a function that generates and returns $N$ random numbers. Make a histogram to examine the distribution of the numbers (the $y_n$ numbers are randomly distributed if the histogram is approximately flat). Name of program file: `diffeq_random.py`. ◇

**Exercise 8.17.** *Make a class for drawing balls from a hat.*

Consider the example about drawing colored balls from a hat in Chapter 8.3.3. It could be handy to have an object that acts as a hat:

```
# make a hat with balls of 3 colors, each color appearing
# on 4 balls:
hat = Hat(colors=('red', 'black', 'blue'), number_of_each_color=4)

# draw 3 balls at random
balls = hat.draw(number_of_balls=3)
```

Realize such code with a class `Hat`. You can borrow useful code from the `balls_in_hat.py` program and ideas from Chapter 8.2.5. Use the `Hat` class to solve the probability problem from Exercise 8.4. Name of program file: `Hat.py`. ◇

**Exercise 8.18.** *Independent vs. dependent random numbers.*

Generate a sequence of $N$ independent random variables with values 0 or 1 and print out this sequence without space between the numbers (i.e., as `001011010110111010`).

The next task is to generate random zeros and ones that are dependent. If the last generated number was 0, the probability of generating a new 0 is $p$ and a new 1 is $1 - p$. Conversely, if the last generated was 1, the probability of generating a new 1 is $p$ and a new 0 is $1 - p$. Since the new value depends on the last one, we say the variables are dependent. Implement this algorithm in a function returning an array of $N$ zeros and ones. Print out this array in the condense format as described above.

Choose $N = 80$ and try $p = 0.5$, $0 = 0.8$ and $p = 0.9$. Can you describe the differences between sequences of independent and dependent random variables? Name of program file: `dependent_random_variables.py`. ◇

**Exercise 8.19.** *Compute the probability of flipping a coin.*

Modify the program from either Exercise 8.1 or 8.13 to incorporate the following extensions: look at a subset $N_1 \leq N$ of the experiments and compute probability of getting a head ($M_1/N_1$, where $M_1$ is the number of heads in $N_1$ experiments). Choose $N = 1000$ and print out

the probability for $N_1 = 10, 100, 500, 1000$. (Generate just $N$ numbers once in the program.) How do you think the accuracy of the computed probability vary with $N_1$? Is the output compatible with this expectation? Name of program file: `flip_coin_prob.py`. ◇

**Exercise 8.20.** *Extend Exer. 8.19.*

We address the same problem as in Exercise 8.19, but now we want to study the probability of getting a head, $p$, as a function of $N_1$, i.e., for $N_1 = 1, \ldots, N$. We also want to vectorize all operations in the code. A first try to compute the probability array for $p$ is

```
h = where(r <= 0.5, 1, 0)
p = zeros(N)
for i in range(N):
    p[i] = sum(h[:i+1])/float(i+1)
```

An array `q[i] = sum(h([:i]))` reflects a *cumulative sum* and can be efficiently generated by the `cumsum` function in `numpy`: `q = cumsum(h)`. Thereafter we can compute `p` by `q/I`, where `I[i]=i+1` and `I` can be computed by `arange(1,N+1)` or `r_[1:N+1]`. Implement both the loop over `i` and the vectorized version based on `cumsum` and check in the program that the resulting `p` array has the same elements (for this purpose you have to compare `float` elements and you can use the `float_eq` function from SciTools, see Exercise 2.51, or the `allclose` function in `numpy` (`float_eq` actually uses `allclose` for array arguments)). Plot `p` against `I` for the case where $N = 10000$. Annotate the axis and the plot with relevant text. Name of program file: `flip_coin_prob_developm.py`. ◇

**Exercise 8.21.** *Simulate the problems in Exer. 3.26.*

Exercise 3.26 describes some problems that can be solved exactly using the formula (3.8), but we can also simulate these problems and find approximate numbers for the probabilities. That is the task of this exercise.

Make a general function `simulate_binomial(p, n, x)` for running $n$ experiments, where each experiment have two outcomes, with probabilities $p$ and $1-p$. The $n$ experiments constitute a "success" if the outcome with probability $p$ occurs exactly $x$ times. The `simulate_binomial` function must repeat the $n$ experiments $N$ times. If $M$ is the number of "successes" in the $N$ experiments, the probability estimate is $M/N$. Let the function return this probability estimate together with the error (the exact result is (3.8)). Simulate the three cases in Exercise 3.26 using this function. Name of program file: `simulate_binomial_problems.py`. ◇

**Exercise 8.22.** *Simulate a poker game.*

Make a program for simulating the development of a poker (or simplified poker) game among $n$ players. Use ideas from Chapter 8.2.4. Name of program file: `poker.py`. ◇

**Exercise 8.23.** *Write a non-vectorized version of a code.*

Read the file `birth_policy.py` containing the code from Chapter 8.3.4. To prove that you understand what is going on in this simulation, replace all the vectorized code by explicit loops over the random arrays. For such code it is natural to use Python's standard `random` module instead of `numpy.random`. However, to verify your alternative implementation it is convenient to have the same sequence of random numbers in the two programs. Therefore, use `numpy`'s `random` module, but use it like the standard Python `random` module, i.e., draw real numbers one at a time instead of a whole array at once. Name of program file: `birth_policy2.py`. ◇

**Exercise 8.24.** *Estimate growth in a simulation model.*

The simulation model in Chapter 8.3.4 predicts the number of individuals from generation to generation. Make a simulation of the "one son" policy with 10 generations, a male portion of 0.51 among newborn babies, set the fertility to 0.92, and assume that 6% of the population will break the law and want 6 children in a family. These parameters implies a significant growth of the population. See if you can find a factor $r$ such that the number of individuals in generation $n$ fulfills the difference equation

$$x_n = (1 + r)x_{n-1} \,.$$

Hint: Compute $r$ for two consecutive generations $x_{n-1}$ and $x_n$ ($r = x_n/x_{n-1} - 1$) and see if $r$ is approximately constant through the evolution of the generations. Name of program file: `growth_birth_policy.py`.
◇

**Exercise 8.25.** *Investigate guessing strategies for Ch. 8.4.1.*

In the game from Chapter 8.4.1 it is smart to use the feedback from the program to track an interval $[p, q]$ that must contain the secret number. Start with $p = 1$ and $q = 100$. If the user guesses at some number $n$, update $p$ to $n + 1$ if $n$ is less than the secret number (no need to care about numbers smaller than $n + 1$), or update $q$ to $n - 1$ if $n$ is larger than the secret number (no need to care about numbers larger than $n - 1$).

Are there any smart strategies to pick a new guess $s \in [p, q]$? To answer this question, investigate two possible strategies: $s$ as the midpoint in the interval $[p, q]$, or $s$ as a uniformly distributed random integer in $[p, q]$. Make a program that implements both strategies, i.e., the player is not prompted for a guess but the computer computes the guess based on the chosen strategy. Let the program run a large number of games and see if either of the strategies can be considered as superior in the long run. Name of program file: `strategies4guess.py`. ◇

**Exercise 8.26.** *Make a vectorized solution to Exer. 8.7.*

Vectorize the simulation program from Exercise 8.7 with the aid of the module `numpy.random` and the `numpy.sum` function. Name of program file: `sum9_4dice_vec.py`. ◇

**Exercise 8.27.** *Compare two playing strategies.*

Suggest a player strategy for the game in Chapter 8.4.2. Remove the question in the `player_guess` function in the file `src/random/ndice2.py`, and implement the chosen strategy instead. Let the program play a large number of games, and record the number of times the computer wins. Which strategy is best in the long run: the computer's or yours? Name of program file: `simulate_strategies1.py`. ◇

**Exercise 8.28.** *Solve Exercise 8.27 with different no. of dice.*

Solve Exercise 8.27 for two other cases with 3 and 50 dice, respectively. Name of program file: `simulate_strategies2.py`. ◇

**Exercise 8.29.** *Extend Exercise 8.28.*

Extend the program from Exercise 8.28 such that the computer and the player can use a different number of dice. Let the computer choose a random number of dice between 2 and 20. Experiment to find out if there is a favorable number of dice for the player. Name of program file: `simulate_strategies3.py`. ◇

**Exercise 8.30.** *Compute $\pi$ by a Monte Carlo method.*

Use the method in Chapter 8.5.2 to compute $\pi$ by computing the area of a circle. Choose $G$ as the circle with its center at the origin and with unit radius, and choose $B$ as the rectangle $[-1,1]\times[-1,1]$. A point $(x,y)$ lies within $G$ if $x^2+y^2<1$. Compare the approximate $\pi$ with `math.pi`. Name of program file: `MC_pi.py`. ◇

**Exercise 8.31.** *Do a variant of Exer. 8.30.*

This exercise has the same purpose of computing $\pi$ as in Exercise 8.30, but this time you should choose $G$ as a circle with center at $(2,1)$ and radius 4. Select an appropriate rectangle $B$. A point $(x,y)$ lies within a circle with center at $(x_c,y_c)$ and with radius $R$ if $(x-x_c)^2+(y-y_c)^2<R^2$. Name of program file: `MC_pi2.py`. ◇

**Exercise 8.32.** *Compute $\pi$ by a random sum.*

Let $x_0,\ldots,x_N$ be $N+1$ uniformly distributed random numbers between 0 and 1. Explain why the random sum $S_N=\sum_{i=0}^{N}2(1-x_i^2)^{-1}$ is an approximation to $\pi$. (Hint: Interpret the sum as Monte Carlo integration and compute the corresponding integral exactly by hand.) Make a program for plotting $S_N$ versus $N$ for $N=10^k$, $k=0,1/2,1,3/2,2,5/2,\ldots,6$. Write out the difference between $S_{10^6}$ and `pi` from the `math` module. Name of program file: `MC_pi_plot.py`. ◇

**Exercise 8.33.** *1D random walk with drift.*

Modify the `walk1D.py` program such that the probability of going to the right is $r$ and the probability of going to the left is $1-r$ (draw numbers in $[0,1)$ rather than integers in $\{1,2\}$). Compute the average position of $n_p$ particles after 100 steps, where $n_p$ is read from the command line. Mathematically one can show that the average position approaches $rn_s - (1-r)n_s$ as $n_p \to \infty$. Write out this exact result together with the computed mean position with a finite number of particles. Name of program file: `walk1D_drift.py`. ◇

**Exercise 8.34.** *1D random walk until a point is hit.*

Set `np=1` in the `walk1Dv.py` program and modify the program to measure how many steps it takes for one particle to reach a given point $x = x_p$. Give $x_p$ on the command line. Report results for $x_p = 5, 50, 5000, 50000$. Name of program file: `walk1Dv_hit_point.py`. ◇

**Exercise 8.35.** *Make a class for 2D random walk.*

The purpose of this exercise is to reimplement the `walk2D.py` program from Chapter 8.7.1 with the aid of classes. Make a class `Particle` with the coordinates $(x, y)$ and the time step number of a particle as attributes. A method `move` moves the particle in one of the four directions and updates the $(x, y)$ coordinates. Another class, `Particles`, holds a list of `Particle` objects and a `plotstep` parameter (as in `walk2D.py`). A method `move` moves all the particles one step, a method `plot` can make a plot of all particles, while a method `moves` performes a loop over time steps and calls `move` and `plot` in each step.

Equip the `Particle` and `Particles` classes with print functionality such that one can print out all particles in a nice way by saying `print p` (for a `Particles` instance `p`) or `print self` (inside a method). Hint: In `__str__`, apply the `pformat` function from the `pprint` module to the list of particles, and make sure that `__repr__` just reuse `__str__` in both classes.

To verify the implementation, print the first three positions of four particles in the `walk2D.py` program and compare with the corresponding results produced by the class-based implementation (the seed of the random number generator must of course be fixed identically in the two programs). You can just perform `p.move()` and `print p` three times in a `verify` function to do this verification task.

Organize the complete code as a module such that the classes `Particle` and `Particles` can be reused in other programs. The test block should call a `run(N)` method to run the walk for `N` steps, where `N` is given on the command line.

Compare the efficiency of the class version against the vectorized version in `walk2Dv.py`, using the techniques of Appendix E.6.1. Name of program file: `walk2Dc.py`. ◇

**Exercise 8.36.** *Vectorize the class code from Exer. 8.35.*

The program developed in Exercise 8.35 cannot be vectorized as long as we base the implementation on class `Particle`. However, if we remove that class and focus on class `Particles`, the latter can employ arrays for holding the positions of all particles and vectorized updates of these positions in the `moves` method. Use ideas from the `walk2Dv.py` program to vectorize class `Particle`. Verify the code against `walk2Dv.py` as explained in Exercise 8.35, and measure the efficiency gain over the version with class `Particle`. Name of program file: `walk2Dcv.py`. ◇

**Exercise 8.37.** *2D random walk with walls; scalar version.*

Modify the `walk2D.py` program or the `walk2Dc.py` program from Exercise 8.35 so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$. Do not move the particle if the new position of a particle is outside $A$. Name of program file: `walk2D_barrier.py`. ◇

**Exercise 8.38.** *2D random walk with walls; vectorized version.*

Modify the `walk2Dv.py` program so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$. Hint: First perform the moves of one direction. Then test if new positions are outside $A$. Such a test returns a boolean array that can be used as index in the position arrays to pick out the indices of the particles that have moved outside $A$. With this array index, one can move all particles outside $A$ back to the relevant boundary of $A$. Name of program file: `walk2Dv_barrier.py`. ◇

**Exercise 8.39.** *Simulate the mixture of gas molecules.*

Suppose we have a box with a wall dividing the box into two equally sized parts. In one part we have a gas where the molecules are uniformly distributed in a random fashion. At $t = 0$ we remove the wall. The gas molecules will now move around and eventually fill the whole box.

This physical process can be simulated by a 2D random walk inside a fixed area $A$ as introduced in Exercises 8.37 and 8.38 (in reality the motion is three-dimensional, but we only simulate the two-dimensional part of it since we already have programs for doing this). Use the program from either Exercises 8.37 or 8.38 to simulate the process for $A = [0, 1] \times [0, 1]$. Initially, place 10000 particles at uniformly distributed random positions in $[0, 1/2] \times [0, 1]$. Then start the random walk and visualize what happens. Simulate for a long time and make a hardcopy of the animation (an animated GIF file, for instance). Is the end result what you would expect? Name of program file: `disorder1.py`.

Molecules tend to move randomly because of collisions and forces between molecules. We do not model collisions between particles in the random walk, but the nature of this walk, with random movements, simulates the effect of collisions. Therefore, the random walk can be used to model molecular motion in many simple cases. In particular, the random walk can be used to investigate how a quite ordered system,

where one gas fills one half of a box, evolves through time to a more disordered system. ◇

**Exercise 8.40.** *Simulate the mixture of gas molecules.*

Solve Exercise 8.39 when the wall dividing the box is not completely removed, but instead we make a small hole in the wall initially. Name of program file: `disorder2.py`. ◇

**Exercise 8.41.** *Guess beer brands.*

You are presented $n$ glasses of beer, each containing a different brand. You are informed that there are $m \geq n$ possible brands in total, and the names of all brands are given. For each glass, you can pay $p$ euros to taste the beer, and if you guess the right brand, you get $q \geq p$ euros back. Suppose you have done this before and experienced that you typically manage to guess the right brand $T$ times out of 100, so that your probability of guessing the right brand is $b = T/100$.

Make a function `simulate(m, n, p, q, b)` for simulating the beer tasting process. Let the function return the amount of money earned and how many correct guesses ($\leq n$) you made. Call `simulate` a large number of times and compute the average earnings and the probability of getting full score in the case $m = n = 4$, $p = 3$, $q = 6$, and $b = 1/m$ (i.e., four glasses with four brands, completely random guessing, and a payback of twice as much as the cost). How much more can you earn from this game if your ability to guess the right brand is better, say $b = 1/2$? Name of program file: `simulate_beer_tasting.py`. ◇

**Exercise 8.42.** *Simulate stock prices.*

A common mathematical model for the evolution of stock prices can be formulated as a difference equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1}\sqrt{\Delta t} r_{n-1}, \tag{8.16}$$

where $x_n$ is the stock price at time $t_n$, $\Delta t$ is the time interval between two time levels ($\Delta t = t_n - t_{n-1}$), $\mu$ is the growth rate of the stock price, $\sigma$ is the volatility of the stock price, and $r_0, \ldots, r_{n-1}$ are normally distributed random numbers with mean zero and unit standard deviation. An initial stock price $x_0$ must be prescribed together with the input data $\mu$, $\sigma$, and $\Delta t$.

We can make a remark that Equation (8.16) is a Forward Euler discretization of a stochastic differential equation for $x(t)$:

$$\frac{dx}{dt} = \mu x + \sigma N(t),$$

where $N(t)$ is a so-called white noise random time series signal. Such equations play a central role in modeling of stock prices.

Make $R$ realizations of (8.16) for $n = 0, \ldots, N$ for $N = 5000$ steps over a time period of $T = 180$ days with a step size $\Delta t = T/N$. Name of program file: `stock_prices.py`. ◇

**Exercise 8.43.** *Compute with option prices in finance.*

In this exercise we are going to consider the pricing of so-called Asian options. An Asian option is a financial contract where the owner earns money when certain market conditions are satisfied.

The contract is specified by a *strike price* $K$ and a maturity time $T$. It is written on the average price of the underlying stock, and if this average is bigger than the strike $K$, the owner of the option will earn the difference. If, on the other hand, the average becomes less, the owner recieves nothing, and the option matures in the value zero. The average is calculated from the last trading price of the stock for each day.

From the theory of options in finance, the price of the Asian option will be the expected present value of the payoff. We assume the stock price dynamics given as,

$$S(t+1) = (1+r)S(t) + \sigma S(t)\epsilon(t), \tag{8.17}$$

where $r$ is the interest-rate, and $\sigma$ is the volatility of the stock price. The time $t$ is supposed to be measured in days, $t = 0, 1, 2, \ldots$, while $\epsilon(t)$ are independent identically distributed normal random variables with mean zero and unit standard deviation. To find the option price, we must calculate the expectation

$$p = (1+r)^{-T}\mathrm{E}\left[\max\left(\frac{1}{T}\sum_{t=1}^{T} S(t) - K, 0\right)\right]. \tag{8.18}$$

The price is thus given as the expected discounted payoff. We will use Monte Carlo simulations to estimate the expectation. Typically, $r$ and $\sigma$ can be set to $r = 0.0002$ and $\sigma = 0.015$. Assume further $S(0) = 100$.

a) Make a function that simulates a path of $S(t)$, that is, the function computes $S(t)$ for $t = 1, \ldots, T$ for a given $T$ based on the recursive definition in (8.17). The function should return the path as an array.
b) Create a function that finds the average of $S(t)$ from $t = 1$ to $t = T$. Make another function that calculates the price of the Asian option based on $N$ simulated averages. You may choose $T = 100$ days and $K = 102$.
c) Plot the price $p$ as a function of $N$. You may start with $N = 1000$.
d) Plot the error in the price estimation as a function $N$ (assume that the $p$ value corresponding to the largest $N$ value is the "right" price). Try to fit a curve of the form $c/\sqrt{N}$ for some $c$ to this error plot. The purpose is to show that the error is reduced as $1/\sqrt{N}$.

Name of program file: `option_price.py`.

If you wonder where the values for $r$ and $\sigma$ come from, you will find the explanation in the following. A reasonable level for the yearly interest-rate is around 5%, which corresponds to a daily rate $0.05/250 = 0.0002$. The number 250 is chosen because a stock exchange is on average open this amount of days for trading. The value for $\sigma$ is calculated as the volatility of the stock price, corresponding to the standard deviation of the daily returns of the stock defined as $(S(t+1) - S(t))/S(t)$. "Normally", the volatility is around 1.5% a day. Finally, there are theoretical reasons why we assume that the stock price dynamics is driven by $r$, meaning that we consider the *risk-neutral* dynamics of the stock price when pricing options. There is an exciting theory explaining the appearance of $r$ in the dynamics of the stock price. If we want to simulate a stock price dynamics mimicing what we see in the market, $r$ in Equation (8.17) must be substituted with $\mu$, the expected return of the stock. Usually, $\mu$ is higher than $r$. ◇

**Exercise 8.44.** *Compute velocity and acceleration.*

In a laboratory experiment waves are generated through the impact of a model slide into a wave tank. (The intention of the experiment is to model a future tsunami event in a fjord, generated by loose rocks that fall into the fjord.) At a certain location, the elevation of the surface, denoted by $\eta$, is measured at discrete points in time using an ultra-sound wave gauge. The result is a time series of vertical positions of the water surface elevations in meter: $\eta(t_0), \eta(t_1), \eta(t_2), \ldots, \eta(t_n)$. There are 300 observations per second, meaning that the time difference between to neighboring measurement values $\eta(t_i)$ and $\eta(t_{i+1})$ is $h = 1/300$ second.

Write a Python program that accomplishes the following tasks:

1. Read $h$ from the command line.
2. Read the $\eta$ values in the file `src/random/gauge.dat` into an array `eta`.
3. Plot `eta` versus the time values.
4. Compute the velocity $v$ of the surface by the formula

$$v_i \approx \frac{\eta_{i+1} - \eta_{i-1}}{2h}, \quad i = 1, \ldots, n-1\,.$$

   Plot $v$ versus time values in a separate plot.
5. Compute the acceleration $a$ of the surface by the formula

$$a_i \approx \frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \ldots, n-1\,.$$

   Plot $a$ versus the time values in a separate plot.

Name of program file: `labstunami1.py`. ◇

**Exercise 8.45.** *Numerical differentiation of noisy signals.*

The purpose of this exercise is to look into numerical differentiation of time series signals that contain measurement errors. This insight might be helpful when analyzing the noise in real data from a laboratory experiment in Exercises 8.44 and 8.46.

1. Compute a signal
$$\bar{\eta}_i = A\sin(\frac{2\pi}{T}t_i), \quad t_i = i\frac{T}{40}, \; i = 0, \ldots, 200\,.$$
Display $\bar{\eta}_i$ versus time $t_i$ in a plot. Choose $A = 1$ and $T = 2\pi$. Store the $\bar{\eta}$ values in an array `etabar`.
2. Compute a signal with random noise $E_i$,
$$\eta_i = \bar{\eta}_i + E_i,$$
$E_i$ is drawn from the normal distribution with mean zero and standard deviation $\sigma = 0.04A$. Plot this $\eta_i$ signal as circles in the same plot as $\eta_i$. Store the $E_i$ in an array `E` for later use.
3. Compute the first derivative of $\bar{\eta}_i$ by the formula
$$\frac{\bar{\eta}_{i+1} - \bar{\eta}_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$
and store the values in an array `detabar`. Display the graph.
4. Compute the first derivative of the error term by the formula
$$\frac{E_{i+1} - E_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$
and store the values in an array `dE`. Calculate the mean and the standard deviation of `dE`.
5. Plot `detabar` and `detabar + dE`. Use the result of the standard deviation calculations to explain the qualitative features of the graphs.
6. The second derivative of a time signal $\eta_i$ can be computed by
$$\frac{\eta_{i+1} - 2\eta_i + \eta i - 1}{h^2}, \quad i = 1, \ldots, n-1\,.$$
Use this formula on the `etabar` data and save the result in `d2etabar`. Also apply the formula to the `E` data and save the result in `d2E`. Plot `d2etabar` and `d2etabar + d2E`. Compute the standard deviation of `d2E` and compare with the standard deviation of `dE` and `E`. Discuss the plot in light of these standard deviations.

Name of program file: `sine_noise.py`. ◇

**Exercise 8.46.** *Model the noise in the data in Exer. 8.44.*

We assume that the measured data can be modeled as a smooth time signal $\bar{\eta}(t)$ plus a random variation $E(t)$. Computing the velocity of $\eta = \bar{\eta} + E$ results in a smooth velocity from the $\bar{\eta}$ term and a noisy

signal from the $E$ term. We can estimate the level of noise in the first derivative of $E$ as follows. The random numbers $E(t_i)$ are assumed to be independent and normally distributed with mean zero and standard deviation $\sigma$. It can then be shown that

$$\frac{E_{i+1} - E_{i-1}}{2h}$$

produces numbers that come from a normal distribution with mean zero and standard deviation $2^{-1/2}h^{-1}\sigma$. How much is the original noise, reflected by $\sigma$, magnified when we use this numerical approximation of the velocity?

The fraction

$$\frac{E_{i+1} - 2E_i + E_{i-1}}{h^2}$$

will also generate numbers from a normal distribution with mean zero, but this time with standard deviation $2h^{-2}\sigma$. Find out how much the noise is magnified in the computed acceleration signal.

The numbers in the `gauge.dat` file are given with 5 digits. This is no certain indication of the accuracy of the measurements, but as a test we may assume $\sigma$ is of the order $10^{-4}$. Check if the visual results for the velocity and acceleration are consistent with the standard deviation of the noise in these signals as modeled above. ◇

**Exercise 8.47.** *Reduce the noise in Exer. 8.44.*

If we have a noisy signal $\eta_i$, where $i = 0, \ldots, n$ counts time levels, the noise can be reduced by computing a new signal where the value at a point is a weighted average of the values at that point and the neighboring points at each side. More precisely, given the signal $\eta_i$, $i = 0, \ldots, n$, we compute a filtered (averaged) signal with values $\eta_i^{(1)}$ by the formula

$$\eta_i^{(1)} = \frac{1}{4}(\eta_{i+1} + 2\eta_i + \eta_{i-1}), \quad i = 1, \ldots, n-1,\ \eta_0^{(1)} = \eta_0,\ \eta_n^{(1)} = \eta_n\,. \tag{8.19}$$

Make a function `filter` that takes the $\eta_i$ values in an array `eta` as input and returns the filtered $\eta_i^{(1)}$ values in an array. Let $\eta_i^{(k)}$ be the signal arising by applying the `filtered` function $k$ times to the same signal. Make a plot with curves $\eta_i$ and the filtered $\eta_i^{(k)}$ values for $k = 1, 10, 100$. Make similar plots for the velocity and acceleration where these are made from both the original $\eta$ data and the filtered data. Discuss the results. Name of program file: `labstunami2.py`. ◇

**Exercise 8.48.** *Find the expected waiting time in traffic lights.*

A driver must pass 10 traffic lights on a certain route. Each light has a period red–yellow-green-yellow of two minutes, of which the green and yellow lights last for 70 seconds. Suppose the driver arrives at a traffic light at some uniformly distributed random point of time during

the period of two minutes. Compute the corresponding waiting time. Repeat this for 10 traffic lights. Run a large number of routes (i.e., repetitions of passing 10 traffic lights) and let the program write out the average waiting time. Does the computed time coincide with what you would expect? Name of program file: `waiting_time.py`. ◇

# 9 Object-Oriented Programming

This chapter introduces the basic ideas of object-oriented programming. Different people put different meanings into the term object-oriented programming: Some use the term for programming with objects in general, while others use the term for programming with class hierarchies. The author applies the second meaning, which is the most widely accepted one in computer science. The first meaning is better named *object-based* programming. Since everything in Python is an object, we do object-based programming all the time, yet one usually reserves this term for the case when classes different from Python's basic types (`int`, `float`, `str`, `list`, `tuple`, `dict`) are involved.

A necessary background for the present chapter is Chapter 7. For Chapters 9.2 and 9.3 one must know basic methods for numerical differentiation and integration, for example from Appendix A. Chapter 9.4 requires knowledge of numerical solution of ordinary differential equations, which is treated in Appendix B and Chapter 7.4. It takes time to grasp the ideas of object-oriented programming, but it will hopefully become clear through many examples. During the initial readings of the chapter, it can be beneficial to skip the more advanced material in Chapters 9.2.3–9.2.6 and 9.4.7–9.4.9.

All the programs associated with this chapter are found in the `src/oo` folder.

## 9.1 Inheritance and Class Hierarchies

Most of this chapter tells you how to put related classes together in families such that the family can be viewed as one unit. This idea helps to hide details in a program, and makes it easier to modify or extend the program.

A family of classes is known as a *class hierarchy*. As in a biological family, there are parent classes and child classes. Child classes can *inherit* data and methods from parent classes, they can modify these data and methods, and they can add their own data and methods. This means that if we have a class with some functionality, we can extend this class by creating a child class and simply add the functionality we need. The original class is still available and the separate child class is small, since it does not need to repeat the code in the parent class.

The magic of object-oriented programming is that other parts of the code do not need to distinguish whether an object is the parent or the child – all generations in a family tree can be treated as a unified object. In other words, one piece of code can work with all members in a class family or hierarchy. This principle has revolutionized the development of large computer systems[1].

The concepts of classes and object-oriented programming first appeared in the Simula programming language in the 1960s. Simula was invented by the Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard, and the impact of the language is particularly evident in C++, Java, and C#, three of the most dominating programming languages in the world today. The invention of object-oriented programming was a remarkable achievement, and the professors Dahl and Nygaard recieved two very prestigious prizes: the von Neumann medal and the Turing prize (popularly known as the Nobel prize of computer science).

A parent class is usually called *base class* or *superclass*, while the child class is known as a *subclass* or *derived class*. We shall use the terms superclass and subclass from now on.

### 9.1.1 A Class for Straight Lines

Assume that we have written a class for straight lines, $y = c_0 + c_1 x$:

```
class Line:
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

[1] Two of the most widely used computer languages today are Java and C#. Both of them force programs to be written in an object-oriented style.

The constructor `__init__` initializes the coefficients $c_0$ and $c_1$ in the expression for the straight line: $y = c_0 + c_1 x$. The call operator `__call__` evaluates the function $c_1 x + c_0$, while the `table` method samples the function at `n` points and creates a table of $x$ and $y$ values.

### 9.1.2 A First Try on a Class for Parabolas

A parabola $y = c_0 + c_1 x + c_2 x^2$ contains a straight line as a special case ($c_2 = 0$). A class for parabolas will therefore be similar to a class for straight lines. All we have do to is to add the new term $c_2 x^2$ in the function evaluation and store $c_2$ in the constructor:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Return a table with n points for L <= x <= R."""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

Observe that we can copy the `table` method from class `Line` without any modifications.

### 9.1.3 A Class for Parabolas Using Inheritance

Python and other languages that support object-oriented programming have a special construct, so that class `Parabola` does not need to repeat the code that we have already written in class `Line`. We can specify that class `Parabola` *inherits* all code from class `Line` by adding "`(Line)`" in the class headline:

```
class Parabola(Line):
```

Class `Parabola` now automatically gets all the code from class `Line` – invisibly. Exercise 9.1 asks you to explicitly demonstrate the validity of this assertion. We say that class `Parabola` is *derived* from class `Line`, or equivalently, that class `Parabola` is a subclass of its superclass `Line`.

Now, class `Parabola` should not be identical to class `Line`: it needs to add data in the constructor (for the new term) and to modify the call operator (because of the new term), but the `table` method can be inherited as it is. If we implement the constructor and the call operator

in class `Parabola`, these will *override* the inherited versions from class `Line`. If we do not implement a `table` method, the one inherited from class `Line` is available as if it were coded visibly in class `Parabola`.

Class `Parabola` must first have the statements from the class `Line` methods `__call__` and `__init__`, and then add extra code in these methods. An important principle in computer programming is to avoid repeating code. We should therefore call up functionality in class `Line` instead of copying statements from class `Line` methods to `Parabola` methods. Any method in the superclass `Line` can be called using the syntax

```
Line.methodname(self, arg1, arg2, ...)
# or
super(Line, self).methodname(arg1, arg2, ...)
```

Let us now show how to write class `Parabola` as a subclass of class `Line`, and implement just the new additional code that we need and that is not already written in the superclass:

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)  # let Line store c0 and c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

This short implementation of class `Parabola` provides exactly the same functionality as the first version of class `Parabola` that we showed on page 481 and that did not inherit from class `Line`. Figure 9.1 shows the class hiearchy in UML fashion. The arrow from one class to another indicates inheritance.

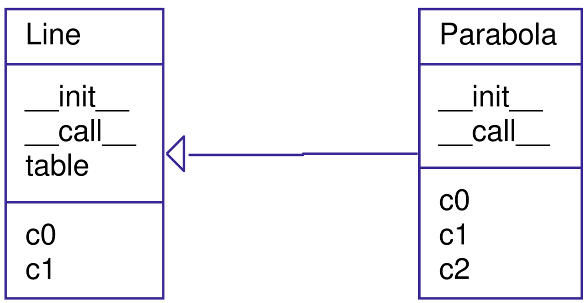



**Fig. 9.1** UML diagram for the class hierarchy with superclass `Line` and subclass `Parabola`.

A quick demo of the `Parabola` class in a main program,

```
p = Parabola(1, -2, 2)
p1 = p(x=2.5)
print p1
print p.table(0, 1, 3)
```

gives this output:

```
8.5
              0           1
            0.5         0.5
              1           1
```

*Program Flow.* The program flow can be somewhat complicated when we work with class hierarchies. Consider the code segment

```
p = Parabola(1, -1, 2)
p1 = p(x=2.5)
```

Let us explain the program flow in detail for these two statements. As always, you can monitor the program flow in a debugger as explained in Chapter D.1.

Calling `Parabola(1, -1, 2)` leads to a call to the constructor method `__init__`, where the arguments `c0`, `c1`, and `c2` in this case are `int` objects with values 1, -1, and 2. The `self` argument in the constructor is the object that will be returned and referred to by the variable `p`. Inside the constructor in class `Parabola` we call the constructor in class `Line`. In this latter method, we create two attributes in the `self` object. Printing out `dir(self)` will explicitly demonstrate what `self` contains so far in the construction process. Back in class `Parabola`'s constructor, we add a third attribute `c2` to the same `self` object. Then the `self` object is invisibly returned and referred to by `p`.

The other statement, `p1 = p(x=2.5)`, has a similar program flow. First we enter the `p.__call__` method with `self` as `p` and `x` as a `float` object with value 2.5. The program flow jumps to the `__call__` method in class `Line` for evaluating the linear part $c_1x + c_0$ of the expression for the parabola, and then the flow jumps back to the `__call__` method in class `Parabola` where we add the new quadratic term.

### 9.1.4 Checking the Class Type

Python has the function `isinstance(i,t)` for checking if an instance `i` is of class type `t`:

```
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
True
>>> isinstance(l, Parabola)
False
```

A `Line` is not a `Parabola`, but is a `Parabola` a `Line`?

```
>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
True
>>> isinstance(p, Line)
True
```

Yes, from a class hierarchy perspective, a `Parabola` instance is regarded as a `Line` instance too, since it contains everything that a `Line` instance contains.

Every instance has an attribute `__class__` that holds the type of class:

```
>>> p.__class__
<class __main__.Parabola at 0xb68f108c>
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__   # string version of the class name
'Parabola'
```

Note that `p.__class__` is a `class` object (or class definition one may say[2]), while `p.__class__.__name__` is a string. These two variables can be used as an alternative test for the class type:

```
if p.__class__.__name__ == 'Parabola':
    <statements>
# or
if p.__class__ == Parabola:
    <statements>
```

However, `isinstance(p, Parabola)` is the recommended programming style for checking the type of an object.

A function `issubclass(c1, c2)` tests if class `c1` is a subclass of class `c2`, e.g.,

```
>>> issubclass(Parabola, Line)
True
>>> issubclass(Line, Parabola)
False
```

The superclasses of a class are stored as a tuple in the `__bases__` attribute of the class object:

```
>>> p.__class__.__bases__
(<class __main__.Line at 0xb7c5d2fc>,)
>>> p.__class__.__bases__[0].__name__  # extract name as string
'Line'
```

### 9.1.5 Attribute versus Inheritance

Instead of letting class `Parabola` inherit from a class `Line`, we may let it *contain* a class `Line` instance as an attribute:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.line = Line(c0, c1)  # let Line store c0 and c1
        self.c2 = c2
```

[2] This means that even the definition of a class, i.e., the class code, is an object that can be referred to by a variable. This is useful in many occasions, see pages 504 and 519.

```
    def __call__(self, x):
        return self.line(x) + self.c2*x**2
```

Whether to use inheritance or an attribute depends on the problem being solved. If it is natural to say that class `Parabola` *is* a `Line` object, we say that `Parabola` has an *is-a relationship* with class `Line`. Alternatively, if it is natural to think that class `Parabola` *has a* `Line` object, we speak about a *has-a relationship* with class `Line`. In the present example, the is-a relationship is most natural since a special case of a parabola is a straight line.

### 9.1.6 Extending versus Restricting Functionality

In our example of `Parabola` as a subclass of `Line`, we used inheritance to *extend* the functionality of the superclass. Inheritance can also be used for *restricting* functionality. Say we have class `Parabola`:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        ...
```

We can define `Line` as a subclass of `Parabola` and restrict the functionality:

```
class Line(Parabola):
    def __init__(self, c0, c1):
        Parabola.__init__(self, c0, c1, 0)
```

The `__call__` and `table` methods can be inherited as they are defined in class `Parabola`.

From this example it becomes clear that there is no unique way of arranging classes in hierarchies. Rather than starting with `Line` and introducing `Parabola`, `Cubic`, and perhaps eventually a general `Polynomial` class, we can start with a general `Polynomial` class and let `Parabola` be a subclass which restricts all coefficients except the first three to be zero. Class `Line` can then be a subclass of `Parabola`, restricting the value of one more coefficient. Exercise 9.4 asks you to implement such a class hierarchy, and to discuss what kind of hierarchy design you like best.

### 9.1.7 Superclass for Defining an Interface

As another example of class hierarchies, we now want to represent functions by classes, as described in Chapter 7.1.2, but in addition to the `__call__` method, we also want to provide methods for the first and second derivative. The class can be sketched as

```
class SomeFunc:
    def __init__(self, parameter1, parameter2, ...)
        # store parameters
    def __call__(self, x):
        # evaluate function
    def df(self, x):
        # evaluate the first derivative
    def ddf(self, x):
        # evaluate the second derivative
```

For a given function, the analytical expressions for first and second derivative must be manually coded. However, we could think of inheriting general functions for computing these derivatives numerically, such that the only thing we must always implement is the function itself. To realize this idea, we create a superclass[3]

```
class FuncWithDerivatives:
    def __init__(self, h=1.0E-9):
        self.h = h  # spacing for numerical derivatives

    def __call__(self, x):
        raise NotImplementedError\
        ('___call__ missing in class %s' % self.__class__.__name__)

    def df(self, x):
        # compute first derivative by a finite difference:
        h = self.h
        return (self(x+h) - self(x-h))/(2.0*h)

    def ddf(self, x):
        # compute second derivative by a finite difference:
        h = self.h
        return (self(x+h) - 2*self(x) + self(x-h))/(float(h)**2)
```

This class is only meant as a superclass of other classes. For a particular function, say $f(x) = \cos(ax) + x^3$, we represent it by a subclass:

```
class MyFunc(FuncWithDerivatives):
    def __init__(self, a):
        self.a = a

    def __call__(self, x):
        return cos(self.a*x) + x**3

    def df(self, x):
        a = self.a
        return -a*sin(a*x) + 3*x**2

    def ddf(self, x):
        a = self.a
        return -a*a*cos(a*x) + 6*x
```

[3] Observe that we carefully ensure that the divisions in methods `df` and `ddf` can never be integer divisions.

The superclass constructor is never called, hence `h` is never initialized, and there are no possibilities for using numerical approximations via the superclass methods `df` and `ddf`. Instead, we override all the inherited methods and implement our own versions. Many think it is a good programming style to always call the superclass constructur in a subclass constructor, even in simple classes where we do not need the functionality of the superclass constructur.

For a more complicated function, e.g., $f(x) = \ln|p\tanh(qx\cos rx)|$, we may skip the analytical derivation of the derivatives, and just code $f(x)$ and rely on the difference approximations inherited from the superclass to compute the derivatives:

```
class MyComplicatedFunc(FuncWithDerivatives):
    def __init__(self, p, q, r, h=1.0E-9):
        FuncWithDerivatives.__init__(self, h)
        self.p, self.q, self.r = p, q, r

    def __call__(self, x):
        return log(abs(self.p*tanh(self.q*x*cos(self.r*x))))
```

That's it! We are now ready to use this class:

```
>>> f = MyComplicatedFunc(1, 1, 1)
>>> x = pi/2
>>> f(x)
-36.880306514638988
>>> f.df(x)
-60.593693618216086
>>> f.ddf(x)
3.3217246931444789e+19
```

Class `MyComplicatedFunc` inherits the `df` and `ddf` methods from the superclass `FuncWithDerivatives`. These methods compute the first and second derivatives approximately, provided that we have defined a `__call__` method. If we fail to define this method, we will inherit `__call__` from the superclass, which just raises an exception, saying that the method is not properly implemented in class `MyComplicatedFunc`.

The important message in this subsection is that we introduced a super class to mainly define an *interface*, i.e., the operations (in terms of methods) that one can do with a class in this class hierarchy. The superclass itself is of no direct use, since it does not implement any function evaluation in the `__call__` method. However, it stores a variable common to all subclasses (`h`), and it implements general methods `df` and `ddf` that any subclass can make use of. A specific mathematical function must be represented as a subclass, where the programmer can decide whether analytical derivatives are to be used, or if the more lazy approach of inheriting general functionality (`df` and `ddf`) for computing numerical derivatives is satisfactory.

In object-oriented programming, the superclass very often defines an interface, and instances of the superclass have no applications on their own – only instances of subclasses can do anything useful.

To digest the present material on inheritance, we recommend to do Exercises 9.1–9.4 before reading the next section.

## 9.2 Class Hierarchy for Numerical Differentiation

Chapter 7.3.2 presents a class `Derivative` that "can differentiate" any mathematical function represented by a callable Python object. The class employs the simplest possible numerical derivative. There are a lot of other numerical formulas for computing approximations to $f'(x)$:

$$f'(x) = \frac{f(x+h)-f(x)}{h} + \mathcal{O}(h), \quad \text{(1st-order forward diff.)} \tag{9.1}$$

$$f'(x) = \frac{f(x)-f(x-h)}{h} + \mathcal{O}(h), \quad \text{(1st-order backward diff.)} \tag{9.2}$$

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + \mathcal{O}(h^2), \quad \text{(2nd-order central diff.)} \tag{9.3}$$

$$f'(x) = \frac{4}{3}\frac{f(x+h)-f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h)-f(x-2h)}{4h} + \mathcal{O}(h^4), \quad \text{(4th-order central diff.)} \tag{9.4}$$

$$f'(x) = \frac{3}{2}\frac{f(x+h)-f(x-h)}{2h} - \frac{3}{5}\frac{f(x+2h)-f(x-2h)}{4h} + \frac{1}{10}\frac{f(x+3h)-f(x-3h)}{6h} + \mathcal{O}(h^6), \quad \text{(6th-order central diff.)} \tag{9.5}$$

$$f'(x) = \frac{1}{h}\left(-\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h)\right) + \mathcal{O}(h^3), \quad \text{(3rd-order forward diff.)} \tag{9.6}$$

The key ideas about the implementation of such a family of formulas are explained in Chapter 9.2.1. For the interested reader, Chapters 9.2.3–9.2.6 contains more advanced additional material that can well be skipped in a first reading. However, the additional material puts the basic solution in Chapter 9.2.1 into a wider perspective, which may increase the understanding of object orientation.

### 9.2.1 Classes for Differentiation

It is argued in Chapter 7.3.2 that it is wise to implement a numerical differentiation formula as a class where $f(x)$ and $h$ are attributes and a `__call__` method makes class instances behave as ordinary Python

functions. Hence, when we have a collection of different numerical differentiation formulas, like (9.1)–(9.6), it makes sense to implement each one of them as a class.

Doing this implementation (see Exercise 7.15), we realize that the constructors are identical because their task in the present case to store $f$ and $h$. Object-orientation is now a natural next step: We can avoid duplicating the constructors by letting all the classes inherit the common constructor code. To this end, we introduce a superclass `Diff` and implement the different numerical differentiation rules in subclasses of `Diff`. Since the subclasses inherit their constructor, all they have to do is to provide a `__call__` method that implements the relevant differentiation formula.

Let us show what the superclass `Diff` looks like and how three subclasses implement the formulas (9.1)–(9.3):

```
class Diff:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Backward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

These small classes demonstrates an important feature of object-orientation: code common to many different classes are placed in a superclass, and the subclasses add just the code that differs among the classes.

We can easily implement the formulas (9.4)–(9.6) by following the same method:

```
class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h)   - f(x-h))  /(2*h) - \
               (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)

class Central6(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (3./2) *(f(x+h)   - f(x-h))  /(2*h) - \
               (3./5) *(f(x+2*h) - f(x-2*h))/(4*h) + \
               (1./10)*(f(x+3*h) - f(x-3*h))/(6*h)

class Forward3(Diff):
    def __call__(self, x):
```

```
        f, h = self.f, self.h
        return (-(1./6)*f(x+2*h) + f(x+h) - 0.5*f(x) - \
                (1./3)*f(x-h))/h
```

Here is a short example of using one of these classes to numerically differentiate the sine function[4]:

```
>>> from Diff import *
>>> from math import sin
>>> mycos = Central4(sin)
>>> # compute sin'(pi):
>>> mycos(pi)
-1.000000082740371
```

Instead of a plain Python function we may use an object with a `__call__` method, here exemplified through the function $f(t; a, b, c) = at^2 + bt + c$:

```
class Poly2:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def __call__(self, t):
        return self.a*t**2 + self.b*t + self.c

f = Poly2(1, 0, 1)
dfdt = Central4(f)
t = 2
print "f'(%g)=%g" % (t, dfdt(t))
```

Let us examine the program flow. When Python encounters `dfdt = Central4(f)`, it looks for the constructor in class `Central4`, but there is no constructor in that class. Python then examines the superclasses of `Central4`, listed in `Central4.__bases__`. The superclass `Diff` contains a constructor, and this method is called. When Python meets the `dfdt(t)` call, it looks for `__call__` in class `Central4` and finds it, so there is no need to examine the superclass. This process of looking up methods of a class is called *dynamic binding.*

*Computer Science Remark.* Dynamic binding means that a name is bound to a function while the program is running. Normally, in computer languages, a function name is static in the sense that it is hardcoded as part of the function body and will not change during the execution of the program. This principle is known as static binding of function/method names. Object orientation offers the technical means to associate different functions with the same name, which yields a kind of magic for incrased flexibility in programs. The particular function that the name refers to can be set at run-time, i.e., when the program is running, and therefore known as dynamic binding.

In Python, dynamic binding is a natural feature since names (variables) can refer to functions and therefore be dynamically bound dur-

[4] We have placed all the classes in the file `Diff.py` such that these classes constitute a module. In an interactive session or a small program, we must import the differentiation classes from the `Diff` module.

ing execution, just as any ordinary variable. To illustrate this point, let `func1` and `func2` be two Python functions of one argument, and consider the code

```
if input == 'func1':
    f = func1
elif input == 'func2':
    f = func2
y = f(x)
```

Here, the name `f` is bound to one of the `func1` and `func2` function objects while the program is running. This is a result of two features: (i) dynamic typing (so the contents of `f` can change), and (ii) functions being ordinary objects. The bottom line is that dynamic binding comes natural in Python, while it appears more like convenient magic in languages like C++, Java, and C#.

### 9.2.2 A Flexible Main Program

As a demonstration of the power of Python programming, we shall now write a program that accepts a function on the command-line, together with information about the difference type (centered, backward, or forward), the order of the approximation, and a value of the independent variable. The output from the program is the derivative of the given function. An example of the usage of the program goes like this:

Terminal

```
differentiate.py 'exp(sin(x))' Central 2 3.1
-1.04155573055
```

Here, we asked the program to differentiate $f(x) = e^{\sin x}$ at $x = 3.1$ with a central scheme of order 2 (using the `Central2` class in the `Diff` hierarchy).

We can provide any expression with `x` as input and request any scheme from the `Diff` hierarchy, and the derivative will be (approximately) computed. One great thing with Python is that the code is very short:

```
import sys
from Diff import *
from math import *
from scitools.StringFunction import StringFunction

formula = sys.argv[1]
f = StringFunction(formula)
difftype = sys.argv[2]
difforder = sys.argv[3]
classname = difftype + difforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print df(x)
```

Read the code line by line, and convince yourself that you understand what is going on. You may need to review Chapters 3.1.2 and 3.1.4.

One disadvantage is that the code above is limited to `x` as the name of the independent variable. If we allow a 5th command-line argument with the name of the independent variable, we can pass this name on to the `StringFunction` constructor, and suddenly our program works with any name for the independent variable!

```
varname = sys.argv[5]
f = StringFunction(formula, independent_variables=varname)
```

Of course, the program crashes if we do not provide five command-line arguments, and the program does not work properly if we are not careful with ordering of the command-line arguments. There is some way to go before the program is really user friendly, but that is beyond the scope of this chapter.

There are two strengths of the `differentiate.py` program: i) interactive specification of the function and the differentiation method, and ii) identical syntax for calling any differentiation method. With one line we create the subclass instance based on input strings. Many other popular programming languages (C++, Java, C#) cannot perform the `eval` operation while the program is running. The result is that we need `if` tests to turn the input string information into creation of subclass instances. Such type of code would look like this in Python:

```
if classname == 'Forward1':
    df = Forward1(f)
elif classname == 'Backward1':
    df = Backward1(f)
...
```

and so forth. This piece of code is very common in object-oriented systems and often put in a function that is referred to as a *factory function*. Factory functions can be made very compact in Python thanks to `eval`.

### 9.2.3 Extensions

The great advantage of sharing code via inheritance becomes obvious when we want to extend the functionality of a class hierarchy. It is possible to do this by adding more code to the superclass only. Suppose we want to be able to assess the accuracy of the numerical approximation to the derivative by comparing with the exact derivative, if available. All we need to do is to allow an extra argument in the constructor and provide an additional superclass method that computes the error in the numerical derivative. We may add this code to class `Diff`, or we may add it in a subclass `Diff2` and let the other classes for various

numerical differentiation formulas inherit from class `Diff2`. We follow the latter approach:

```
class Diff2(Diff):
    def __init__(self, f, h=1E-9, dfdx_exact=None):
        Diff.__init__(self, f, h)
        self.exact = dfdx_exact

    def error(self, x):
        if self.exact is not None:
            df_numerical = self(x)
            df_exact = self.exact(x)
            return df_exact - df_numerical

class Forward1(Diff2):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

The other subclasses, `Backward1`, `Central2`, and so on, must also be derived from `Diff2` to equip all subclasses with new functionality for perfectly assessing the accuracy of the approximation. No other modifications are necessary in this example, since all the subclasses can inherit the superclass constructor and the `error` method. Figure 9.2 shows a UML diagram of the new `Diff` class hierarchy.

Here is an example of usage:

```
mycos = Forward1(sin, dfdx_exact=cos)
print 'Error in derivative is', mycos.error(x=pi)
```

The program flow of the `mycos.error(x=pi)` call can be interesting to follow. We first enter the `error` method in class `Diff2`, which then calls `self(x)`, i.e., the `__call__` method in class `Forward1`, which jumps out to the `self.f` function, i.e., the `sin` function in the `math` module in the present case. After returning to the `error` method, the next call is to `self.exact`, which is the `cos` function (from `math`) in our case.

*Application.* We can apply the methods in the `Diff2` hierarchy to get some insight into the accuracy of various difference formulas. Let us write out a table where the rows correspond to different $h$ values, and the columns correspond to different approximation methods (except the first column which reflects the $h$ value). The values in the table can be the numerically computed $f'(x)$ or the error in this approximation if the exact derivative is known. The following function writes such a table:

```
def table(f, x, h_values, methods, dfdx=None):
    # print headline (h and class names for the methods):
    print '      h       ',
    for method in methods:
        print '%-15s' % method.__name__,
    print  # newline
    for h in h_values:
        print '%10.2E' % h,
        for method in methods:
```

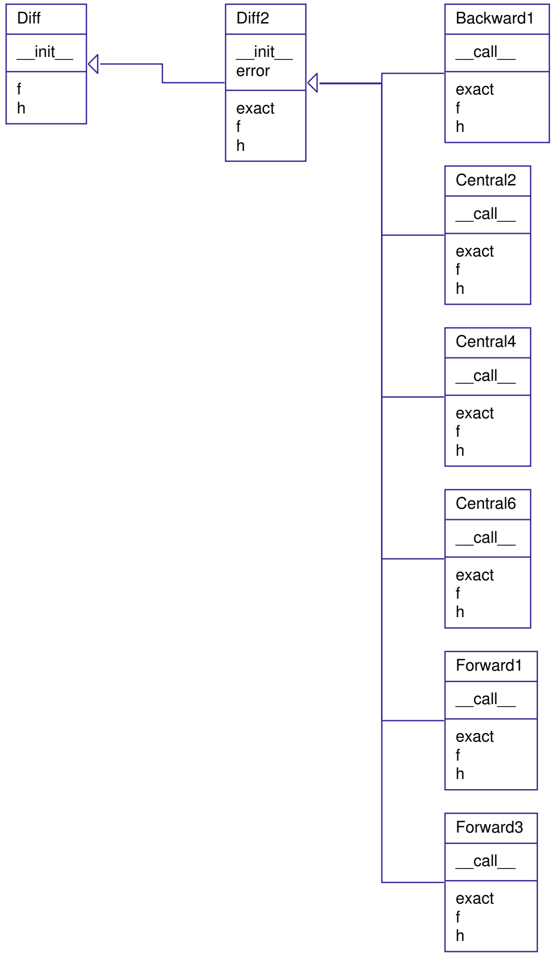



**Fig. 9.2** UML diagram of the `Diff` hierarchy for a series of differentiation formulas (`Backward1`, `Central2`, etc.).

```
                if dfdx is not None:        # write error
                    d = method(f, h, dfdx)
                    output = d.error(x)
                else:                       # write value
                    d = method(f, h)
                    output = d(x)
                print '%15.8E' % output,
            print  # newline
```

The next lines tries three approximation methods on $f(x) = e^{-10x}$ for $x = 0$ and with $h = 1, 1/2, 1/4, 1/16, \ldots, 1/512$:

```
from Diff2 import *
from math import exp

def f1(x):
    return exp(-10*x)

def df1dx(x):
    return -10*exp(-10*x)
```

```
table(f1, 0, [2**(-k) for k in range(10)],
      [Forward1, Central2, Central4], df1dx)
```

Note how convenient it is to make a list of class names – class names can be used as ordinary variables, and to print the class name as a string we just use the `__name__` attribute. The output of the main program above becomes

```
     h         Forward1         Central2         Central4
1.00E+00 -9.00004540E+00  1.10032329E+04 -4.04157586E+07
5.00E-01 -8.01347589E+00  1.38406421E+02 -3.48320240E+03
2.50E-01 -6.32833999E+00  1.42008179E+01 -2.72010498E+01
1.25E-01 -4.29203837E+00  2.81535264E+00 -9.79802452E-01
6.25E-02 -2.56418286E+00  6.63876231E-01 -5.32825724E-02
3.12E-02 -1.41170013E+00  1.63556996E-01 -3.21608292E-03
1.56E-02 -7.42100948E-01  4.07398036E-02 -1.99260429E-04
7.81E-03 -3.80648092E-01  1.01756309E-02 -1.24266603E-05
3.91E-03 -1.92794011E-01  2.54332554E-03 -7.76243120E-07
1.95E-03 -9.70235594E-02  6.35795004E-04 -4.85085874E-08
```

From one row to the next, $h$ is halved, and from about the 5th row and onwards, the `Forward1` errors are also halved, which is consistent with the error $\mathcal{O}(h)$ of this method. Looking at the 2nd column, we see that the errors are reduced to 1/4 when going from one row to the next, at least after the 5th row. This is also according to the theory since the error is proportional to $h^2$. For the last row with a 4th-order scheme, the error is reduced by 1/16, which again is what we expect when the error term is $\mathcal{O}(h^4)$. What is also interesting to observe, is the benefit of using a higher-order scheme like `Central4`: with, for example, $h = 1/128$ the `Forward1` scheme gives an error of $-0.7$, `Central2` improves this to 0.04, while `Central4` has an error of $-0.0002$. More accurate formulas definitely give better results[5]. The test example shown here is found in the file `Diff2_examples.py`.

### 9.2.4 Alternative Implementation via Functions

Could we implement the functionality offered by the `Diff` hierarchy of objects by using plain functions and no object orientation? The answer is "yes, almost". What we have to pay for a pure function-based solution is a less friendly user interface to the differentiation functionality: More arguments must be supplied in function calls, because each difference formula, now coded as a straight Python function, must get $f(x)$, $x$, and $h$ as arguments. In the class version we first store $f$ and $h$ as attributes in the constructor, and every time we want to compute the derivative, we just supply $x$ as argument.

A Python function for implementing numerical differentiation reads

[5] Strictly speaking, it is the fraction of the work and the accuracy that counts: `Central4` needs four function evaluations, while `Central2` and `Forward1` only needs two.

```
def central2_func(f, x, h=1.0E-9):
    return (f(x+h) - f(x-h))/(2*h)
```

The usage demonstrates the difference from the class solution:

```
mycos = central2_func(sin, pi, 1E-6)
# compute sin'(pi):
print "g'(%g)=%g (exact value is %g)" % (pi, mycos, cos(pi))
```

Now, `mycos` is a number, not a callable object. The nice thing with the class solution is that `mycos` appeared to be a standard Python function whose mathematical values equal the derivative of the Python function `sin(x)`. But does it matter whether `mycos` is a function or a number? Yes, it matters if we want to apply the difference formula twice to compute the second-order derivative. When `mycos` is a callable object of type `Central2`, we just write

```
mysin = Central2(mycos)
# or
mysin = Central2(Central2(sin))

# compute g''(pi):
print "g''(%g)=%g" % (pi, mysin(pi))
```

With the `central2_func` function, this composition will not work. Moreover, when the derivative is an object, we can send this object to any algorithm that expects a mathematical function, and such algorithms include numerical integration, differentation, interpolation, ordinary differential equation solvers, and finding zeros of equations, so the applications are many.

### 9.2.5 Alternative Implementation via Functional Programming

As a conclusion of the previous section, the great benefit of the object-oriented solution in Chapter 9.2.1 is that one can have some subclass instance `d` from the `Diff` (or `Diff2`) hieararchy and write `d(x)` to evaluate the derivative at a point `x`. The `d(x)` call behaves as if `d` were a standard Python function containing a manually coded expression for the derivative.

The `d(x)` interface to the derivative can also be obtained by other and perhaps more direct means than object-oriented programming. In programming languages where functions are ordinary objects that can be referred to by variables, as in Python, one can make a function that returns the right `d(x)` function according to the chosen numerical derivation rule. The code looks as this:

```
def differentiate(f, method, h=1.0E-9):
    h = float(h)  # avoid integer division

    if method == 'Forward1':
```

```
        def Forward1(x):
            return (f(x+h) - f(x))/h
        return Forward1

    elif method == 'Backward1':
        def Backward1(x):
            return (f(x) - f(x-h))/h
        return Backward1
    ...
```

And the usage is like this:

```
mycos = differentiate(sin,   'Forward1')
mysin = differentiate(mycos, 'Forward1')
x = pi
print mycos(x), cos(x), mysin, -sin(x)
```

The surprising thing is that when we call `mycos(x)` we provide only `x`, while the function itself looks like

```
def Forward1(x):
    return (f(x+h) - f(x))/h
return Forward1
```

How do the parameters `f` and `h` get their values when we call `mycos(x)`? There is some magic attached to the `Forward1` function, or literally, there are some variables attached to `Forward1`: this function "remembers" the values of `f` and `h` that existed as local variables in the `differentiate` function when the `Forward1` function was defined.

In computer science terms, the `Forward1` always has access to variables in the *scope* in which the function was defined. The `Forward1` function is what is known as a *closure* in some computer languages. Closures are much used in a programming style called *functional programming*. Two key features of functional programming is operations on lists (like list comprehensions) and returning functions from functions. Python supports functional programming, but we will not consider this programming style further in this book.

### 9.2.6 Alternative Implementation via a Single Class

Instead of making many classes or functions for the many different differentiation schemes, the basic information about the schemes can be stored in one table. With a single method in one single class can use the table information, and for a given scheme, compute the derivative. To do this, we need to reformulate the mathematical problem (actually by using ideas from Chapter 9.3.1).

A family of numerical differentiation schemes can be written

$$f'(x) \approx \sum_{i=-r}^{r} w_i f(x_i), \qquad (9.7)$$

where $w_i$ are weights and $x_i$ are points. The $2r+1$ points are symmetric around some point $x$:

$$x_i = x + ih, \quad i = -r, \ldots, r\,.$$

The weights depend on the differentation scheme. For example, the midpoint scheme (9.3) has

$$w_{-1} = -1, \quad w_0 = 0, \quad w_1 = 1\,.$$

Table 9.1 lists the values of $w_i$ for different difference formulas. In this table we have set $r = 4$, which is sufficient for the schemes written up in this book.

Given a table of the $w_i$ values, we can use (9.7) to compute the derivative. A faster, vectorized computation can have the $x_i$, $w_i$, and $f(x_i)$ values as stored in three vectors. Then $\sum_i w_i f(x_i)$ can be interpreted as a dot product between the two vectors with components $w_i$ and $f(x_i)$, respectively.

**Table 9.1** Weights in some difference schemes. The number after the nature of a scheme denotes the order of the schemes (for example, "central 2" is a central difference of 2nd order).

| points | $x-4h$ | $x-3h$ | $x-2h$ | $x-h$ | $x$ | $x+h$ | $x+2h$ | $x+3h$ | $x+4h$ |
|---|---|---|---|---|---|---|---|---|---|
| central 2 | 0 | 0 | 0 | $-\frac{1}{2}$ | 0 | $\frac{1}{2}$ | 0 | 0 | 0 |
| central 4 | 0 | 0 | $\frac{1}{12}$ | $-\frac{2}{3}$ | 0 | $\frac{2}{3}$ | $-\frac{1}{12}$ | 0 | 0 |
| central 6 | 0 | $-\frac{1}{60}$ | $\frac{3}{20}$ | $-\frac{3}{4}$ | 0 | $\frac{3}{4}$ | $-\frac{3}{20}$ | $\frac{1}{60}$ | 0 |
| central 8 | $\frac{1}{280}$ | $-\frac{4}{105}$ | $\frac{12}{60}$ | $-\frac{4}{5}$ | 0 | $\frac{4}{5}$ | $-\frac{12}{60}$ | $\frac{4}{105}$ | $-\frac{1}{280}$ |
| forward 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| forward 3 | 0 | 0 | 0 | $-\frac{2}{6}$ | $-\frac{1}{2}$ | 1 | $-\frac{1}{6}$ | 0 | 0 |
| backward 1 | 0 | 0 | 0 | $-1$ | 1 | 0 | 0 | 0 | 0 |

A class with the table of weights as a static variable, a constructor, and a `__call__` method for evaluating the derivative via $\sum_i w_i f(x_i)$ looks as follows:

```
class Diff3:
    table = {
    ('forward', 1):
    [0, 0, 0, 0, 1, 1, 0, 0, 0],
    ('central', 2):
    [0, 0, 0, -1./2, 0, 1./2, 0, 0, 0],
    ('central', 4):
    [ 0, 0, 1./12, -2./3, 0, 2./3, -1./12, 0, 0],
    ...
    }
    def __init__(self, f, h=1.0E-9, type='central', order=2):
        self.f, self.h, self.type, self.order = f, h, type, order
        self.weights = array(Diff2.table[(type, order)])

    def __call__(self, x):
        f_values = array([f(self.x+i*self.h) for i in range(-4,5)])
        return dot(self.weights, f_values)/self.h
```

Here we used numpy's dot(x, y) function for computing the inner or dot product between two arrays x and y.

Class Diff3 can be found in the file Diff3.py. Using class Diff3 to differentiate the sine function goes like this:

```
import Diff3
mycos = Diff3.Diff3(sin, type='central', order=4)
print "sin'(pi):", mycos(pi)
```

*Remark.* The downside of class Diff3, compared with the other implementation techniques, is that the sum $\sum_i w_i f(x_i)$ contains many multiplications by zero for lower-order schemes. These multiplications are known to yield zero in advance so we waste computer resources on trivial calculations. Once upon a time, programmers would have been extremely careful to avoid wasting multiplications this way, but today arithmetic operations are quite cheap, especially compared to fetching data from the computer's memory. Lots of other factors also influence the computational efficiency of a program, but this is beyond the scope of this book.

## 9.3 Class Hierarchy for Numerical Integration

There are many different numerical methods for integrating a mathematical function, just as there are many different methods for differentiating a function. It is thus obvious that the idea of object-oriented programming and class hierarchies can be applied to numerical integration formulas in the same manner as we did in Chapter 9.2.

### 9.3.1 Numerical Integration Methods

First, we list some different methods for integrating $\int_a^b f(x)dx$ using $n$ evaluation points. All the methods can be written as

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i), \tag{9.8}$$

where $w_i$ are weights and $x_i$ are evaluation points, $i = 0, \ldots, n-1$. The Midpoint method has

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h, \quad h = \frac{b-a}{n}, \quad i = 0, \ldots, n-1\,. \tag{9.9}$$

The Trapezoidal method has the points

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}, \quad i = 0, \ldots, n-1, \tag{9.10}$$

and the weights

$$w_0 = w_{n-1} = \frac{h}{2}, \ w_i = h, \quad i = 1, \dots, n-2\,. \tag{9.11}$$

Simpson's rule has the same evaluation points as the Trapezoidal rule, but

$$h = 2\frac{b-a}{n-1}, \quad w_0 = w_{n-1} = \frac{h}{6}, \tag{9.12}$$

$$w_i = \frac{h}{3} \text{ for } i = 2, 4, \dots, n-3, \tag{9.13}$$

$$w_i = \frac{2h}{3} \text{ for } i = 1, 3, 5, \dots, n-2\,. \tag{9.14}$$

Note that $n$ must be odd in Simpson's rule. A Two-Point Gauss-Legendre method takes the form

$$x_i = a + (i + \frac{1}{2})h - \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 0, 2, 4, \dots, n-2, \tag{9.15}$$

$$x_i = a + (i + \frac{1}{2})h + \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 1, 3, 5, \dots, n-1, \tag{9.16}$$

with $h = 2(b-a)/n$. Here $n$ must be even. All the weights have the same value: $w_i = h/2$, $i = 0, \dots, n-1$. Figure 9.3 illustrates how the points in various integration rules are distributed over a few intervals.

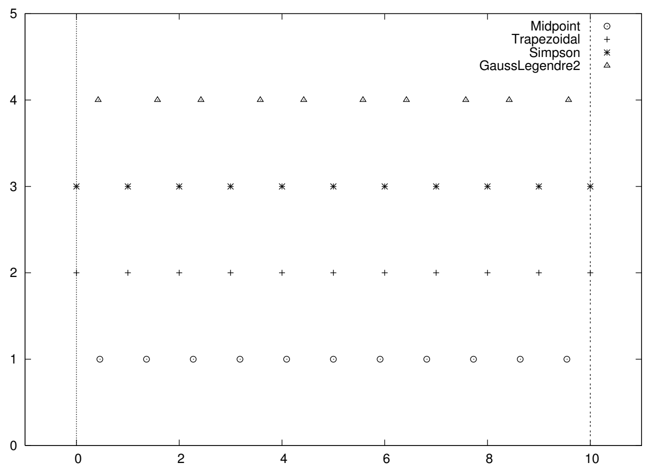



**Fig. 9.3** Illustration of the distribution of points for various numerical integration methods. The Gauss-Legendre method has 10 points, while the other methods have 11 points in $[0, 10]$.

### 9.3.2 Classes for Integration

We may store $x_i$ and $w_i$ in two NumPy arrays and compute the integral as $\sum_{i=0}^{n-1} w_i f(x_i)$. This operation can also be vectorized as a dot (inner) product between the $w_i$ vector and the $f(x_i)$ vector, provided $f(x)$ is implemented in a vectorizable form.

We argued in Chapter 7.3.3 that it pays off to implement a numerical integration formula as a class. If we do so with the different methods from the previous section, a typical class looks like this:

```
class SomeIntegrationMethod:
    def __init__(self, a, b, n):
        # compute self.points and self.weights

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

Making such classes for many different integration methods soon reveals that all the classes contain common code, namely the `integrate` method for computing $\sum_{i=0}^{n-1} w_i f(x_i)$. Therefore, this common code can be placed in a superclass, and subclasses can just add the code that is specific to a certain numerical integration formula, namely the definition of the weights $w_i$ and the points $x_i$.

Let us start with the superclass:

```
class Integrator:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.points, self.weights = self.construct_method()

    def construct_method(self):
        raise NotImplementedError('no rule in class %s' % \
                                  self.__class__.__name__)

    def integrate(self, f):
        s = 0
        for i in range(len(self.weights)):
            s += self.weights[i]*f(self.points[i])
        return s
```

As we have seen, we store the $a$, $b$, and $n$ data about the integration method in the constructor. Moreover, we compute arrays or lists `self.points` for the $x_i$ points and `self.weights` for the $w_i$ weights. All this code can now be inherited by all subclasses.

The initialization of points and weights is put in a seperate method, `construct_method`, which is supposed to be implemented in each subclass, but the superclass provides a default implementation which tells the user that the method is not implemented. What happens is that when subclasses redefine a method, that method overrides the method inherited from the superclass. Hence, if we forget to redefine `construct_method` in a subclass, we will inherit the one from the superclass, and this method issues an error message. The construction of

this error message is quite clever in the sense that it will tell in which class the `construct_method` method is missing (`self` will be the subclass instance and its `__class__.__name__` is a string with the corresponding subclass name).

In computer science one usually speaks about *overloading* a method in a subclass, but the words redefining and overriding are also used. A method that is overloaded is said to be *polymorphic*. A related term, *polymorphism*, refers to coding with polymorphic methods. Very often, a superclass provides some default implementation of a method, and a subclass overloads the method with the purpose of tailoring the method to a particular application.

The `integrate` method is common for all integration rules, i.e., for all subclasses, so it can be inherited as it is. A vectorized version can also be added in the superclass to make it automatically available also in all subclasses:

```
    def vectorized_integrate(self, f):
        return dot(self.weights, f(self.points))
```

Let us then implement a subclass. Only the `construct_method` method needs to be written. For the Midpoint rule, this is a matter of translating the formulas in (9.9) to Python:

```
class Midpoint(Integrator):
    def construct_method(self):
        a, b, n = self.a, self.b, self.n  # quick forms
        h = (b-a)/float(n)
        x = linspace(a + 0.5*h, b - 0.5*h, n)
        w = zeros(len(x)) + h
        return x, w
```

Observe that we implemented directly a vectorized code. We could also have used (slow) loops and explicit indexing:

```
        x = zeros(n)
        w = zeros(n)
        for i in range(n):
            x[i] = a + 0.5*h + i*h
            w[i] = h
```

Before we continue with other subclasses for other numerical integration formulas, we will have a look at the program flow when we use class `Midpoint`. Suppose we want to integrate $\int_0^2 x^2 dx$ using 101 points:

```
def f(x): return x*x
m = Midpoint(0, 2, 101)
print m.integrate(f)
```

How is the program flow? The assignment to `m` invokes the constructor in class `Midpoint`. Since this class has no constructor, we invoke the inherited one from the superclass `Integrator`. Here attributes are

stored, and then the `construct_method` method is called. Since `self` is a `Midpoint` instance, it is the `construct_method` in the `Midpoint` class that is invoked, even if there is a method with the same name in the superclass. Class `Midpoint` overloads `construct_method` in the superclass. In a way, we "jump down" from the constructor in class `Integrator` to the `construct_method` in the `Midpoint` class. The next statment, `m.integrate(f)`, just calls the inherited `integral` method that is common to all subclasses.

A vectorized Trapezoidal rule can be implemented in another subclass with name `Trapezoidal`:

```
class Trapezoidal(Integrator):
    def construct_method(self):
        x = linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)
        w = zeros(len(x)) + h
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

Observe how we divide the first and last weight by 2, using index 0 (the first) and -1 (the last) and the `/=` operator (`a /= b` is equivalent to `a = a/b`). Here also we could have implemented a scalar version with loops. The relevant code is in function `trapezoidal` in Chapter 7.3.3.

Class Simpson has a slightly more demanding rule, at least if we want to vectorize the expression, since the weights are of two types.

```
class Simpson(Integrator):
    def construct_method(self):
        if self.n % 2 != 1:
            print 'n=%d must be odd, 1 is added' % self.n
            self.n += 1
        x = linspace(self.a, self.b, self.n)
        h = (self.b - self.a)/float(self.n - 1)*2
        w = zeros(len(x))
        w[0:self.n:2] = h*1.0/3
        w[1:self.n-1:2] = h*2.0/3
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

We first control that we have an odd number of points, by checking that the remainder of `self.n` divided by two is 1. If not, an execption could be raised, but for smooth operation of the class, we simply increase $n$ so it becomes odd. Such automatic adjustments of input is not a rule to be followed in general. Wrong input is best notified explicitly. However, sometimes it is user friendly to make small adjustments of the input, as we do here, to achieve a smooth and successful operation. (In cases like this, a user might become uncertain whether the answer can be trusted if she (later) understands that the input should not yield a correct result. Therefore, do the adjusted computation, and provide a notification to the user about what has taken place.)

The computation of the weights `w` in class `Simpson` applies slices with stride (jump/step) 2 such that the operation is vectorized for speed. Recall that the upper limit of a slice is not included in the set, so `self.n-1` is the largest index in the first case, and `self.n-2` is the largest index in the second case. Instead of the vectorized operation of slices for computing `w`, we could use (slower) straight loops:

```
        for i in range(0, self.n, 2):
            w[i] = h*1.0/3
        for i in range(1, self.n-1, 2):
            w[i] = h*2.0/3
```

The points in the Two-Point Gauss-Legendre rule are slightly more complicated to calculate, so here we apply straight loops to make a safe first implementation:

```
class GaussLegendre2(Integrator):
    def construct_method(self):
        if self.n % 2 != 0:
            print 'n=%d must be even, 1 is subtracted' % self.n
            self.n -= 1
        nintervals = int(self.n/2.0)
        h = (self.b - self.a)/float(nintervals)
        x = zeros(self.n)
        sqrt3 = 1.0/sqrt(3)
        for i in range(nintervals):
            x[2*i]   = self.a + (i+0.5)*h - 0.5*sqrt3*h
            x[2*i+1] = self.a + (i+0.5)*h + 0.5*sqrt3*h
        w = zeros(len(x)) + h/2.0
        return x, w
```

A vectorized calculation of `x` is possible by observing that the `(i+0.5)*h` expression can be computed by `linspace`, and then we can add the remaining two terms:

```
m = linspace(0.5*h, (nintervals-1+0.5)*h, nintervals)
x[0:self.n-1:2] = m + self.a - 0.5*sqrt3*h
x[1:self.n:2]   = m + self.a + 0.5*sqrt3*h
```

The array on the right-hand side has half the length of `x` ($n/2$), but the length matches exactly the slice with stride 2 on the left-hand side.

### 9.3.3 Using the Class Hierarchy

To verify the implementation, we first try to integrate a linear function. All methods should compute the correct integral value regardless of the number of evaluation points:

```
def f(x):
    return x + 2

a = 2; b = 3; n = 4
for Method in Midpoint, Trapezoidal, Simpson, GaussLegendre2:
    m = Method(a, b, n)
    print m.__class__.__name__, m.integrate(f)
```

Observe how we simply list the class names as a tuple (comma-separated objects), and `Method` will in the `for` loop attain the values `Midpoint`, `Trapezoidal`, and so forth. For example, in the first pass of the loop, `Method(a, b, n)` is identical to `Midpoint(a, b, n)`.

The output of the test above becomes

```
Midpoint 4.5
Trapezoidal 4.5
n=4 must be odd, 1 is added
Simpson 4.5
GaussLegendre2 4.5
```

Since $\int_2^3 (x+2)dx = \frac{9}{2} = 4.5$, all methods passed this simple test.

A more challenging integral, from a numerical point of view, is

$$\int\limits_0^1 \left(1 + \frac{1}{m}\right) t^{\frac{1}{m}} dt = 1\,.$$

To use any subclass in the `Integrator` hierarchy, the integrand must be a function of one variable only. For the present integrand, which depends on $t$ and $m$, we use a class to represent it:

```
class F:
    def __init__(self, m):
        self.m = float(m)

    def __call__(self, t):
        m = self.m
        return (1 + 1/m)*t**(1/m)
```

We now ask the question: How much is the error in the integral reduced as we increase the number of integration points ($n$)? It appears that the error decreases exponentially with $n$, so if we want to plot the errors versus $n$, it is best to plot the logarithm of the error versus $\ln n$. We expect this graph to be a straight line, and the steeper the line is, the faster the error goes to zero as $n$ increases. A common conception is to regard one numerical method as better than another if the error goes faster to zero as we increase the computational work (here $n$).

For a given $m$ and method, the following function computes two lists containing the logarithm of the $n$ values, and the logarithm of the corresponding errors in a series of experiments:

```
def error_vs_n(f, exact, n_values, Method, a, b):
    log_n = []  # log of actual n values (Method may adjust n)
    log_e = []  # log of corresponding errors
    for n_value in n_values:
        method = Method(a, b, n_value)
        error = abs(exact - method.integrate(f))
        log_n.append(log(method.n))
        log_e.append(log(error))
    return log_n, log_e
```

We can plot the error versus $n$ for several methods in the same plot and make one plot for each $m$ value. The loop over $m$ below makes such plots:

```
n_values = [10, 20, 40, 80, 160, 320, 640]
for m in 1./4, 1./8., 2, 4, 16:
    f = F(m)
    figure()
    for Method in Midpoint, Trapezoidal, \
            Simpson, GaussLegendre2:
        n, e = error_vs_n(f, 1, n_values, Method, 0, 1)
        plot(n, e); legend(Method.__name__); hold('on')
    title('m=%g' % m); xlabel('ln(n)'); ylabel('ln(error)')
```

The code snippets above are collected in a function `test` in the `integrate.py` file.

The plots for $m > 1$ look very similar. The plots for $0 < m < 1$ are also similar, but different from the $m > 1$ cases. Let us have a look at the results for $m = 1/4$ and $m = 2$. The first, $m = 1/4$, corresponds to $\int_0^1 5x^4 dx$. Figure 9.4 shows that the error curves for the Trapezoidal and Midpoint methods converge more slowly compared to the error curves for Simpson's rule and the Gauss-Legendre method. This is the usual situation for these methods, and mathematical analysis of the methods can confirm the results in Figure 9.4.

However, when we consider the integral $\int_0^1 \frac{3}{2}\sqrt{x}dx$, $(m = 2)$ and $m > 1$ in general, all the methods converge with the same speed, as shown in Figure 9.5. Our integral is difficult to compute numerically when $m > 1$, and the theoretically better methods (Simpson's rule and the Gauss-Legendre method) do not converge faster than the simpler methods. The difficulty is due to the infinite slope (derivative) of the integrand at $x = 0$.

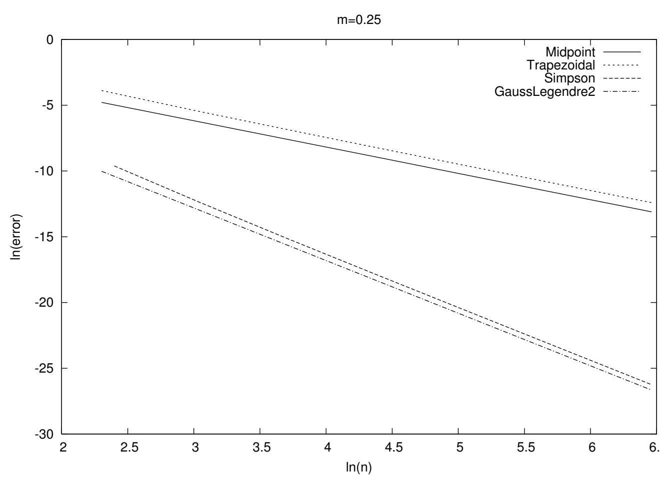



**Fig. 9.4** The logarithm of the error versus the logarithm of integration points for integral $5x^4$ computed by the Trapezoidal and Midpoint methods (upper two lines), and Simpson's rule and the Gauss-Legendre methods (lower two lines).

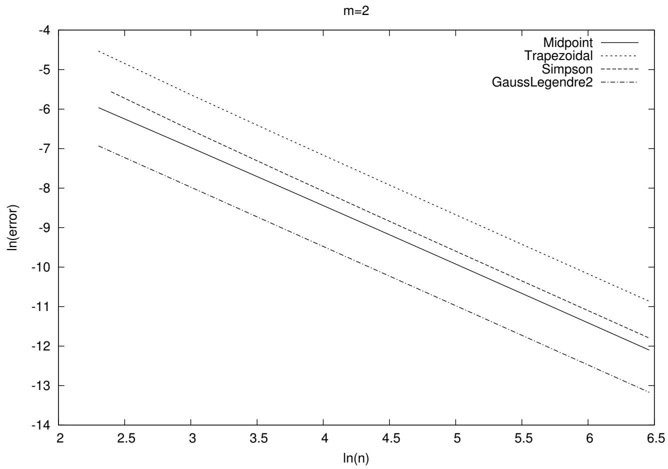



**Fig. 9.5** The logarithm of the error versus the logarithm of integration points for integral $\frac{3}{2}\sqrt{x}$ computed by the Trapezoidal method and Simpson's rule (upper two lines), and Midpoint and Gauss-Legendre methods (lower two lines).

### 9.3.4 About Object-Oriented Programming

From an implementational point of view, the advantage of class hierarchies in Python is that we can save coding by inheriting functionality from a superclass. In programming languages where each variable must be specified with a fixed type, class hierarchies are particularly useful because a function argument with a special type also works with all subclasses of that type. Suppose we have a function where we need to integrate:

```
def do_math(arg1, arg2, integrator):
    ...
    I = integrator.integrate(myfunc)
    ...
```

That is, `integrator` must be an instance of some class, or a module, such that the syntax `integrator.integrate(myfunc)` corresponds to a function call, but nothing more (like having a particular type) is demanded.

This Python code will run as long as `integrator` has a method `integrate` taking one argument. In other languages, the function arguments are specified with a type, say in Java we would write

```
void do_math(double arg1, int arg2, Simpson integrator)
```

A compiler will examine all calls to `do_math` and control that the arguments are of the right type. Instead of specifying the integration method to be of type `Simpson`, one can in Java and other object-oriented languages specify `integrator` to be of the superclass type `Integrator`:

```
void do_math(double arg1, int arg2, Integrator integrator)
```

Now it is allowed to pass an object of any subclass type of `Integrator` as the third argument. That is, this method works with `integrator` of type `Midpoint`, `Trapezoidal`, `Simpson`, etc., not just one of them. Class hierarchies and object-oriented programming are therefore important means for parameterizing away types in languages like Java, C++, and C#. We do not need to parameterize types in Python, since arguments are not declared with a fixed type. Object-oriented programming is hence not so technically important in Python as in other languages for providing increased flexibility in programs.

Is there then any use for object-oriented programming beyond inheritance? The answer is yes! For many code developers object-oriented programming is not just a technical way of sharing code, but it is more a way of modeling the world, and understanding the problem that the program is supposed to solve. In mathematical applications we already have objects, defined by the mathematics, and standard programming concepts such as functions, arrays, lists, and loops are often sufficient for solving simpler problems. In the non-mathematical world the concept of objects is very useful because it helps to structure the problem to be solved. As an example, think of the phone book and message list software in a mobile phone. Class `Person` can be introduced to hold the data about one person in the phone book, while class `Message` can hold data related to an SMS message. Clearly, we need to know who sent a message so a `Message` object will have an associated `Person` object, or just a phone number if the number is not registered in the phone book. Classes help to structure both the problem and the program. The impact of classes and object-oriented programming on modern software development can hardly be exaggerated.

## 9.4 Class Hierarchy for Numerical Methods for ODEs

The next application targets numerical solution of ordinary differential equations. There is a jungle of such solution methods, a fact that suggests collecting the methods in a class hierarchy, just as we did for numerical differentation and integration formulas.

### 9.4.1 Mathematical Problem

We may distinguish between two types of ordinary differential equations (ODEs): *scalar ODEs* and *systems of ODEs*. The former type involves one single equation,

$$\frac{du}{dt} = f(u,t), \quad u(0) = u_0, \tag{9.17}$$

with one unknown function $u(t)$. The latter type involves $n$ equations with $n$ unknown functions $u^{(i)}(t)$, $i = 0, \ldots, n-1$:

$$\frac{du^{(i)}}{dt} = f(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t), \tag{9.18}$$

In addition, we need $n$ initial conditions for a system with $n$ equations and unknowns:

$$u^{(i)}(0) = u_0^{(i)}, \quad i = 0, \ldots, n-1\,. \tag{9.19}$$

It is common to collect the functions $u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}$ in a vector

$$u = (u^{(0)}, u^{(1)}, \ldots, u^{(n-1)})$$

and the initial conditions also in a vector

$$u_0 = (u_0^{(0)}, u_0^{(1)}, \ldots, u_0^{(n-1)})\,.$$

In that case, (9.18) and (9.19) can be written as (9.17). We may take important advantage of this fact in an implementation: If the vectors are represented by Numerical Python arrays, the code we write for a scalar will very often work for arrays too. Unless you are quite familiar with systems of ODEs and array arithmetics, it can be a good idea to just think about scalar ODEs and that $u(t)$ is a function of one variable when you read on. Later, you can come back and reread the text with systems of ODEs and $u(t)$ as a vector (array) in mind. The text that follows and the program code are in fact independent of whether we solve scalar ODEs or systems of ODEs. This is quite a remarkable achievement, obtained by using a clever mathematical notation, where we do not distinguish between $u$ as a scalar function or as a vector of functions, and the fact that scalar and array computations in Python look the same[6].

*Example of a System of ODEs.* An oscillating spring-mass system can be governed by a second-order ODE (see (C.8) in Appendix C for derivation):

$$mu'' + \beta u' + ku = F(t), \quad u(0) = u_0,\ u'(0) = 0\,.$$

The parameters $m$, $\beta$, and $k$ are known and $F(t)$ is a prescribed function. This second-order equation can be rewritten as two first-order equations by introducing two functions (see Chapter B.5),

[6] The invisible difference between scalar ODEs and systems of ODEs is not only important for addressing *both* newcomers to ODEs and more experienced readers. The principle is very important for software development too: We can write code with scalar ODEs in mind and test this code. Afterwards, the code should also work immediately for systems and $u(t)$ as a vector of functions!

$$u^{(0)}(t) = u(t), \quad u^{(1)}(t) = u'(t)\,.$$

The unknowns are now the position $u^{(0)}(t)$ and the velocity $u^{(1)}(t)$. We can then create equations where the derivative of the two new primary unknowns $u^{(0)}$ and $u^{(1)}$ appear alone on the left-hand side:

$$\frac{d}{dt}u^{(0)}(t) = u^{(1)}(t), \tag{9.20}$$

$$\frac{d}{dt}u^{(1)}(t) = m^{-1}(F(t) - \beta u^{(1)} - ku^{(0)})\,. \tag{9.21}$$

It is common to express such a system as $u'(t) = f(u, t)$ where now $u$ and $f$ are vectors, here of length two:

$$u(t) = (u^{(0)}(t), u^{(1)}(t))$$

$$f(t, u) = (u^{(1)}, m^{-1}(F(t) - \beta u^{(1)} - ku^{(0)}))\,. \tag{9.22}$$

### 9.4.2 Numerical Methods

Numerical methods for ODEs compute approximations $u_k$ to $u$ at discrete time levels $t_k$, $k = 1, 2, 3, \ldots$. With a constant time step size $\Delta t$ in time, we have $t_k = k\Delta t$. Some of the simplest, but also most widely used methods for ODEs are listed below.

1. The Forward Euler method:

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)\,. \tag{9.23}$$

2. The Midpoint method:

$$u_{k+1} = u_{k-1} + 2\Delta t f(u_k, t_k), \tag{9.24}$$

   for $k = 1, 2, \ldots$. For the first step, to compute $u_1$, the formula (9.24) involves $u_{-1}$, which is unknown, so here we must use another method, for instance, (9.23).
3. The 2nd-order Runge-Kutta method:

$$u_{k+1} = u_k + K_2 \tag{9.25}$$

   where

$$K_1 = \Delta t\, f(u_k, t_k), \tag{9.26}$$

$$K_2 = \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t)\,. \tag{9.27}$$

4. The 4th-order Runge-Kutta method:

$$u_{k+1} = u_k + \frac{1}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right), \tag{9.28}$$

where

$$K_1 = \Delta t\, f(u_k, t_k), \tag{9.29}$$

$$K_2 = \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t), \tag{9.30}$$

$$K_3 = \Delta t\, f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t), \tag{9.31}$$

$$K_4 = \Delta t\, f(u_k + K3, t_k + \Delta t)\,. \tag{9.32}$$

5. The Backward Euler method:

$$u_{k+1} = u_k + \Delta t\, f(u_{k+1}, t_{k+1})\,. \tag{9.33}$$

If $f(u,t)$ is nonlinear in $u$, (9.33) constitutes a nonlinear equation in $u_{k+1}$, which must be solved by some method for nonlinear equations, say Newton's method (see Chapter 9.4.4 for more details).

The methods above are valid both for scalar ODEs and for systems of ODEs. In the system case, the quantities $u$, $u_k$, $u_{k+1}$, $f$, $K_1$, $K_2$, etc., are vectors.

### 9.4.3 The ODE Solver Class Hierarchy

Chapter 7.4.2 presents a class `ForwardEuler` for implementing the Forward Euler scheme (9.23). Most of the code in this class is independent of the numerical method we use. In fact, we only need to change the `advance` method[7]. if we want another method, such as the 4-th order Runge-Kutta method, instead of the Forward Euler scheme. Copying the `ForwardEuler` class and editing just the `advance` method is considered bad programming practice, because we get two copies the general parts of class `ForwardEuler`. As we implement more schemes, we end up with a lot of copies of the same code. Correcting an error or improving the code in this general part then requires identical edits in several almost identical classes.

A good programming practice is to collect all the common code in a superclass. Subclasses can implement the `advance` method and share all other code.

*The Superclass.* We introduce class `ODESolver` as the superclass of all numerical methods for solving ODEs. Class `ODESolver` should provide all functionality that is common to all numerical methods for ODEs:

1. hold the solution $u(t)$ at discrete time points in a list `u`
2. hold the corresponding time values `t`

[7] For more advanced methods than (9.23)–(9.33), especially so-called adaptive methods where $\Delta t$ is automatically adjusted to gain an overall accuracy of the computations, there are more details that differ between various solution methods.

3. hold information about the $f(u,t)$ function, i.e., a callable Python object `f(u, t)`
4. hold the (constant) time step $\Delta t$ in an attribute `dt`
5. hold the time step number $k$ in an attribute `k`
6. set the initial condition $u_0$
7. implement the loop over all time steps

As already outlined, we implement the last point as two methods: `solve` for performing the time loop and `advance` for advancing the solution one time step. The latter method is empty in the superclass since the method is to be implemented by various subclasses for various numerical schemes.

A first version class `ODESolver` may follow the structure and contents of class `ForwardEuler` from Chapter 7.4.2:

```
class ODESolver:
    def __init__(self, f, dt):
        self.f = f
        self.dt = dt

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError

    def set_initial_condition(self, u0, t0=0):
        self.u = []    # u[k] is solution at time t[k]
        self.t = []    # time levels in the solution process

        self.u.append(float(u0))
        self.t.append(float(t0))
        self.k = 0  # time level counter (k in formulas)

    def solve(self, T):
        """Advance solution in time until t <= T."""
        tnew = 0
        while tnew <= T:
            unew = self.advance()
            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)
```

Provided that `self.advance()` returns the solution at a new time level, this superclass contains everything needed to solve an ODE.

We can make an improvement of the `solve` method as explained in Exercise 7.26: The time loop is run as long as $k \leq N$ *and* a user-defined function `terminate(u,t,k)` is `False`. By default, `terminate` can be the value `False`, and if desired, the programmer supplies some function that can be used to terminate the time loop on basis of the lists `u` and `t` and the time step counter `k`. For example, if we want to solve an ODE until the solution becomes zero, we can supply the function

```
def terminate(u, t, k):
    eps = 1.0E-6           # small number
    if abs(u[-1]) < eps:   # close enough to zero?
        return True
```

```
    else:
        return False
```

The `solve` method then looks like

```
    def solve(self, T, terminate=None):
        """
        Advance solution from t = t0 to t <= T, steps of dt
        as long as terminate(u,t,k) is False.
        terminate(u,t,k) is a user-given function
        returning True or False. By default, a terminate
        function which always returns False is used.
        """
        if terminate is None:
            terminate = lambda u, t, k: False
        self.k = 0
        tnew = 0
        while tnew <= T and \
            not terminate(self.u, self.t, self.k):

            unew = self.advance()

            self.u.append(unew)
            tnew = self.t[-1] + self.dt
            self.t.append(tnew)
            self.k += 1
        return numpy.array(self.u), numpy.array(self.t)
```

We use a default value of `None` to indicate that the user has not provided a `terminate` function. In that case, we make a `terminate` function that always returns `False` (see Chapter 2.2.11 for an explanation of using `lambda` for quickly defining a function).

*The Forward Euler Method.* Subclasses implement specific numerical formulas for numerical solution of ODEs in the `advance` method. For the Forward Euler the formula is given by (9.23). All data we need for this formula are stored as attributes by the superclass. First we load these data into variables with shorter names, to avoid the lengthy `self` prefix and obtain a notation closer to the mathematics. Then we apply the formula (9.23), and finally we return the new value:

```
class ForwardEuler(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        unew = u[k] + dt*f(u[k], t)
        return unew
```

A remark is worth mentioning: When we extract attributes to local variables with short names, we can only use these local variables for reading values, not setting values. For example, if we do a `k += 1` to update the time step counter, that increased value is not reflected in `self.k` (which is the "official" counter). Extracting class attributes in local variables is done for getting the code closer to the mathematics, but has a danger of introducing bugs that might be hard to track down.

*Systems of ODEs.* The codes for solving ODEs have up to now be written scalar ODEs, not systems. However, we can with very small adjustments make all the code work with systems as well. In an ODE system, `f(u[k], t)` returns a list or array, depending on what the user prefers when implementing the right-hand side function. If a list is returned, we face a problem with `dt*f(u[k], t)` since multiplication of a float and a list is not defined. Therefore, we should automatically convert all right-hand sides to arrays. This is tedious to do inside the numerical algorithm. It is better to do it once and for all by redefining `self.f` in the constructor: we let `self.f` be a function that calls the user-given `f` and then feeds the returned list or array to `numpy.asarray` to ensure that we have an array to compute with. The `asarray` function does nothing if the argument is already an array. The following adjustment is then needed in the constructor:

```
def f_wrapper(u, t):
    return numpy.asarray(f(u, t), float)
self.f = f_wrapper

# or just
self.f = lambda u, t: numpy.asarray(f(u, t), float)
```

No other modifications are necessary for the ODE solvers to work perfectly with systems of ODEs, although we had only scalar ODEs in mind when we wrote the code!

*The 4th-order Runge-Kutta Method.* Below is an implementation of the 4th-order Runge-Kutta method (9.28):

```
class RungeKutta4(ODESolver):
    def advance(self):
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]
        dt2 = dt/2.0
        K1 = dt*f(u[k], t)
        K2 = dt*f(u[k] + 0.5*K1, t + dt2)
        K3 = dt*f(u[k] + 0.5*K2, t + dt2)
        K4 = dt*f(u[k] + K3, t + dt)
        unew = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return unew
```

As long as the right-hand side function `f` is guaranteed to return an array in the case we want to solve an ODE system, the implemented Runge-Kutta method works systems of ODEs as well as for scalar ODEs.

It is left as exercises to implement other numerical methods in the `ODESolver` class hierarchy (see Exercises 9.27 and 9.28). However, the Backward Euler method (9.33) requires a much more advanced implementation than the other methods so that particular method deserves its own section.

### 9.4.4 The Backward Euler Method

The Backward Euler scheme (9.33) leads in general to a *nonlinear* equation at a new time level, while all the other schemes listed in Chapter 9.4.2 has a simple formula for a new $u_{k+1}$ value. We see that (9.33) gives an equation to be solved for $u_{k+1}$ by rearranging

$$u_{k+1} = u_k + \Delta t\, f(u_{k+1}, t_{k+1})$$

to

$$F(u_{k+1}) \equiv u_{k+1} - \Delta t f(u_{k+1}, t_{k+1}) - u_k = 0\,.$$

We must solve the equation $F(u_{k+1}) = 0$ with respect to $u_{k+1}$. It may be easier to see this, and later easier to implement method, if we introduce a new variable $w$ for $u_{k+1}$. The equation to be solved is then

$$F(w) \equiv w - \Delta t f(w, t_k) - u_k = 0\,. \tag{9.34}$$

If now $f(u, t)$ is a nonlinear function of $u$, $F(w)$ will also be a nonlinear function of $w$.

To solve $F(w) = 0$ we can use the Bisection method from Chapter 3.6.2, Newton's method from Chapter 5.1.9, or the Secant method from Exercise 5.14. Here we apply Newton's method and the implementation given in `src/diffeq/Newton.py`. A disadvantage with Newton's method is that we need the derivative of $F$ with respect to $w$, which requires the derivative $\partial f(w, t)/\partial w$. A quick solution is to use a numerical derivative. Class `Derivative` from Chapter 7.3.2.

We make a subclass `BackwardEuler`. As we need to solve $F(w) = 0$ at every time step, we also need to implement the $F(w)$ function. We can do this in a method, as in

```
class BackwardEuler:
    def F(self, w):
        return w - \
        self.dt*self.f(w, self.t[-1]) - self.u[self.k]
```

Alternatively, we can make `F` as a local function inside the `advance` method[8]:

```
    def advance(self):
        u, dt, f, k, t = \
           self.u, self.dt, self.f, self.k, self.t[-1]

        def F(w):
            return w - dt*f(w, t) - u[k]

        dFdw = Derivative(F)
        w_start = u[k] + dt*f(u[k], t)
        unew, n, F_value = Newton(F, w_start, dFdw, N=30)
        if n >= 30:
```

[8] The local variables in the `advance` function, e.g., `dt` and `u`, act as “global” variables for the `F` function. Hence, when `F` is sent away to some `Newton` function, `F` remembers the values of `dt`, `f`, `t`, and `u`!.

```
            print "Newton's failed to converge at t=%g "\
                  "(%d iterations)" % (t, n)
        return unew
```

The `F(w)` now looks closer to the mathematics. There are also some other statements that deserve a comment. The derivative $dF/dw$ is computed numerically by a class `Derivative`, which is a slight modification of the similar class in Chapter 7.3.2, because we now want to use a more accurate, centered formula:

```
class Derivative:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

This code is included in the `ODESolver.py` file after class `BackwardEuler`.

The next step is to call Newton's method. For this purpose we need to import the `Newton` function from the `Newton` module. However, this module is not located in the same folder as the `ODESolver` module, since the latter is in `src/oo` while the former is in `src/diffeq`. To tell Python to look for modules in `src/diffeq`, we modify `sys.path` as explained in Chapter 3.5.3:

```
import sys, os
sys.path.insert(0, os.path.join(os.pardir, 'diffeq'))
from Newton import Newton
```

Note that `diffeq` is a subfolder of our parent folder so we specify `../diffeq` rather than the full and possibly complicated path to `diffeq`. The parent folder is available as `os.pardir` ("parent directory"), and `os.path.join` combines folders with the right delimiter (forward slash on Mac/Linux/Unix and backward slash on Windows).

Having the `Newton` function from Chapter 5.1.9 accessible in our `ODESolver.py`, we can make a call and supply our `F` function as the argument `f`, a start value for the iteration, here called `w_start`, as the argument `x`, and the derivative `dFdw` for the argument `dfdx`. We rely on default values for the `epsilon` and `store` arguments, while the maximum number of iterations is set to `N=30`. The program is terminated if it happens that the number of iterations exceeds that value, because then the method has diverged, and we have not been able to compute the next $u_{k+1}$ value.

The starting value for Newton's method must be chosen. As we expect the solution to not change much from one time level to the next, $u_k$ could be a good initial guess. However, we can do better by using a simple Forward Euler step $u_k + \Delta t f(u_k, t_k)$, which is exactly what we do in the `advance` function above.

Since Newton's method always has the danger of converging slowly, it can be interesting to store the number of iterations at each time level as an attribute in the `BackwardEuler` class. We can easily insert extra statement for this purpose:

```
    def advance(self):
        ...
        unew, n, F_value = Newton(F, w_start, dFdw, N=30)
        if k == 0:
            self.Newton_iter = []
        self.Newton_iter.append(n)
        ...
```

Note the need for creating an empty list (at the first call of `advance`) before we can append elements.

There is now one important question to ask: Will the `advance` method work for systems of ODEs? In that case, $F(w)$ is a vector of functions. The implementation of `F` will work when `w` is a vector, because all the quantities involved in the formula are arrays or scalar variables. The `dFdw` instance will compute a numerical derivative of each component of the vector function `dFdw.f` (which is simply our `F` function). The call to the `Newton` function is more critical: It turns out that this function, as the algorithm behind it, works for scalar equations only. Newton's method can quite easily be extended to a system of nonlinear equations, but we do not consider that topic here. Instead we equip class `BackwardEuler` with a constructor that calls the `f` object and controls that the returned value is a `float` and not an array:

```
class BackwardEuler(ODESolver):
    def __init__(self, f, dt):
        ODESolver.__init__(self, f, dt)
        # make a sample call to check that f is a scalar function:
        value = f(1,1)
        if not isinstance(value, (int, float)):
            raise ValueError\
            ('f(u,t) must return float/int, not %s' % type(value))
```

Observe that we must explicitly call the superclass constructor and pass on the arguments `f` and `dt` to achieve the right storage and treatment of these arguments.

Understanding class `BackwardEuler` implies a good understanding of classes in general, a good understanding of numerical methods for ODEs, for numerical differentiation, and for finding roots of functions, and a good understanding on how to combine different code segments from different parts of the book. Therefore, if you have digested class `BackwardEuler`, you have all reasons to believe that you have digested the key topics of this book.

### 9.4.5 Verification

We use the same verification problem as in Chapter 7.4.3, namely a function $u(t)$ that is linear in $t$ and that will be exactly reproduced by any of our schemes. Choosing $u(t) = 0.2t + 3$ with a corresponding $f(u, t) = 0.2 + (u - 0.2t - 3)^5$, we can write the following code for testing the Forward Euler, Runge-Kutta, and Backward Euler methods:

```
def _f1(u, t):
    return 0.2 + (u - _u_solution_f1(t))**5

def _u_solution_f1(t):
    """Exact u(t) corresponding to _f1 above."""
    return 0.2*t + 3

def _verify(f, exact):
    u0 = 3;  dt = 0.4;  T = 2.8
    for Method_class in ForwardEuler, RungeKutta4, BackwardEuler:
        method = Method_class(f, dt)
        method.set_initial_condition(u0)
        u, t = method.solve(T)
        print Method_class.__name__, ':\n', u
    u_exact = exact(t)
    print 'Exact:\n', u_exact
    print 'Backward Euler iterations:', method.Newton_iter

if __name__ == '__main__':
    _verify(_f1, _u_solution_f1)
```

The output shows that all numerical methods provide exact numbers:

```
ForwardEuler:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
RungeKutta4:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
BackwardEuler :
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
Exact:
[ 3.    3.08  3.16  3.24  3.32  3.4   3.48  3.56]
```

This is a good indication that many parts of our code are correct. (For the Backward Euler method, the test is insuffient because the starting value, being the Forward Euler prediction, is exact. Therefore, Newton's method does not need any interations! Changing the start value to `u[k]` results in a linear equation for $w$ and a need for one Newton iteration.)

### 9.4.6 Application 1: $u' = u$

The perhaps simplest of all ODEs, $u' = u$, is our first target problem for the classes in the `ODESolver` hierarchy. The basic part of the application of class `ForwardEuler` goes as follows:

```
from ODESolver import *
from scitools.std import *

def f(u, t):
```

```
    return u

T = 3
dt = 0.1
method = ForwardEuler(f, dt)
method.set_initial_condition(1.0)
u, t = method.solve(N)
plot(t, u)
```

We can easily demonstrate how superior the 4-th order Runge-Kutta method is for this equation when the time step is bigger ($\Delta t = 1$):

```
dt = 1
figure()
for Method_class in ForwardEuler, RungeKutta4:
    method = Method_class(f, dt)
    method.set_initial_condition(1)
    u, t = method.solve(T)
    plot(t, u)
    legend('%s' % method.__name__)
    hold('on')

t = linspace(0, T, 41)  # finer resolution
plot(t, u_exact)
legend('exact')
```

Figure 9.6 shows the plot. The complete program can be found in the file `app1_exp.py`.

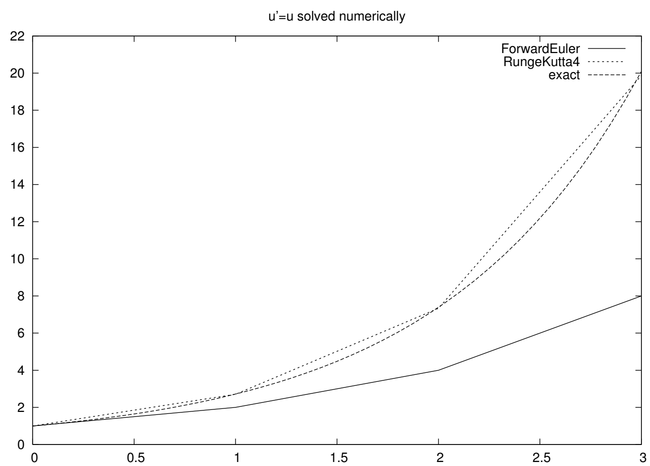



**Fig. 9.6** Comparison of the Forward Euler and the 4-th order Runge-Kutta method for solving $u' = u$ for $t \in [0, 3]$ and a long time step $\Delta t = 1$.

### 9.4.7 Application 2: The Logistic Equation

The logistic ODE (B.23) is copied here for convenience:

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = u_0\,.$$

The right-hand side contains the parameters $\alpha$ and $R$. As emphasized in Chapters 7.1.1–7.1.2, the "right" way to code the right-hand side is then to make a class where $\alpha$ and $R$ are attributes, and where a `__call__` method evaluates the formula for the right-hand side. Such code is explained in Chapter 7.4.4.

However, by a mathematical simplification we can remove the $\alpha$ and $R$ parameters from the ODE and thereby simplify the ODE and also the implementation of the right-hand side. The simplification consists in scaling the independent and dependent variables, which is advantageous to do anyway if the goal is to understand more of the model equation and its solution. The scaling consists in introducing new variables

$$v = \frac{u}{R}, \quad \tau = \alpha t\,.$$

Inserting $u = Rv$ and $t = \tau/\alpha$ in the equation gives

$$\frac{dv}{d\tau} = v(1-v), \quad v(0) = \frac{u_0}{R}\,.$$

Assume that we start with a small population, say $u_0/R = 0.05$. Amazingly, there are no more parameters in the equation for $v(\tau)$. That is, we can solve for $v$ once and for all, and then recover $u(t)$ by

$$u(t) = Rv(\alpha t)\,.$$

Geometrically, the transformation from $v$ to $u$ is just a stretching of the two axis in the coordinate system.

We can compute $v(\tau)$ by the 4-th order Runge-Kutta method in a program:

```
v0 = 0.05
dtau = 0.05
T = 10
method = RungeKutta4(lambda v, tau: v*(1-v), dtau)
method.set_initial_condition(v0)
v, tau = method.solve(T)
```

Observe that we use a lambda function (Chapter 2.2.11) to save some typing of a separate function for the right-hand side of the ODE. Now we need to run the program only once to compute $v(t)$, and from this solution we can easily create the solution $u(t)$, represented in terms of `u` and `t` arrays, by

```
t = alpha*tau
u = R*v
```

Below we make a plot to show how the $u(t)$ curve varies with $\alpha$:

```
def ut(alpha, R):
    return alpha*tau, R*v

figure()
```

```
for alpha in linspace(0.2, 1, 5):
    t, u = ut(alpha, R=1)
    plot(t, u, legend='alpha=%g' % alpha)
    hold('on')
```

The resulting plot appears in Figure 9.7. Without the scaling, we would need to solve the ODE for each desired $\alpha$ value. Futhermore, with the scaling we understand better that the influence of $\alpha$ is only to stretch the $t$ axis, or equivalently, stretch the curve along the $t$ axis.

The complete program for this example is found in the file app2_logistic.py.

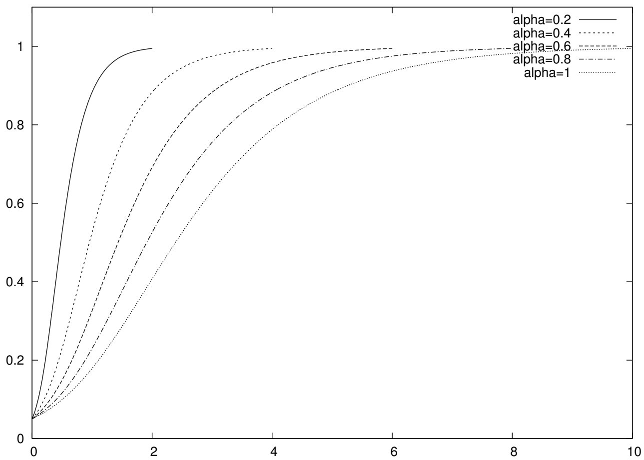



**Fig. 9.7** Solution of the logistic equation $u' = \alpha u \left(1 - \frac{u}{R}\right)$ by the 4-th order Runge-Kutta method for various choices of $\alpha$.

### 9.4.8 Application 3: An Oscillating System

The motion of a box attached to a spring (Appendix C) can be modeled by two first-order differential equations as listed in (9.22) and repeated here for convenience:

$$\frac{du^{(0)}}{dt} = u^{(1)},$$
$$\frac{du^{(1)}}{dt} = w''(t) + g - m^{-1}\beta u^{(1)} - m^{-1}ku^{(0)}\,.$$

We now have a system of two ODEs, and the unknown is a vector containing the two functions, and the right-hand side $f$ is also a vector with two components.

The code related to this example is found in app3_osc.py. Because our right-hand side $f$ contains several parameters, we implement it as a class with the parameters as attributes and a `__call__` method for

returning the 2-vector $f$. We assume that the user of the class supplies the $w(t)$ function, so it is natural to compute $w''(t)$ by a finite difference formula.

```
class OscSystem:
    def __init__(self, m, beta, k, g, w):
        self.m, self.beta, self.k, self.g, self.w = \
                float(m), float(beta), float(k), float(g), w

    def __call__(self, u, t):
        u0, u1 = u
        m, beta, k, g, w = \
           self.m, self.beta, self.k, self.g, self.w
        # use a finite difference for w''(t):
        h = 1E-5
        ddw = (w(t+h) - 2*w(t) + w(t-h))/(2*h)
        f = [u1, ddw  + g - beta/m*u1 - k/m*u0]
        return f
```

A simple test case arises if we set $m = k = 1$ and $\beta = g = w = 0$:

$$\frac{du^{(0)}}{dt} = u^{(1)},$$
$$\frac{du^{(1)}}{dt} = -u^{(0)}\,.$$

Suppose that $u^{(0)}(0) = 1$ and $u^{(1)}(0) = 0$. An exact solution is then

$$u^{(0)}(t) = \cos t, \quad u^{(1)}(t) = -\sin t\,.$$

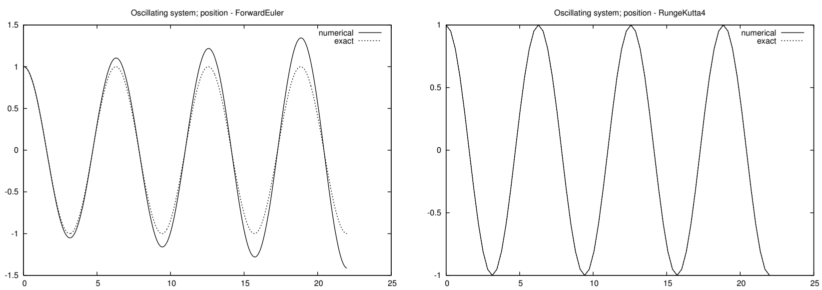
We can use this case to check how the Forward Euler method compares with the 4-th order Runge-Kutta method:

```
f = OscSystem(1.0, 0.0, 1.0, 0.0, lambda t: 0)
u_init = [1, 0]    # initial condition
T = 7*pi
for Method_class in ForwardEuler, RungeKutta4:
    # let ForwardEuler dt be 1/10 of the RungeKutta dt:
    if Method_class == ForwardEuler:
        dt = 2*pi/200
    elif Method_class == RungeKutta4:
        dt = 2*pi/20
    method = Method_class(f, dt)
    method.set_initial_condition(u_init)
    u, t = method.solve(T)

    # u is an array of [u0,u1] pairs for each time level,
    # get the u0 values from u for plotting:
    u0_values = u[:, 0]
    u1_values = u[:, 1]
    u0_exact = cos(t)
    u1_exact = -sin(t)
    figure()
    alg = Method_class.__name__  # (class) name of algorithm
    plot(t, u0_values, 'r-',
         t, u0_exact, 'b-',
         legend=('numerical', 'exact'),
         title='Oscillating system; position - %s' % alg,
         hardcopy='tmp_oscsystem_pos_%s.eps' % alg)
    figure()
    plot(t, u1_values, 'r-',
```

```
           t, u1_exact, 'b-',
           legend=('numerical', 'exact'),
           title='Oscillating system; velocity - %s' % alg,
           hardcopy='tmp_oscsystem_vel_%s.eps' % alg)
```

For this particular application it turns out that the 4-th order Runge-Kutta is very accurate, even with few (20) time steps per oscillation (period). Unfortunately, the Forward Euler method leads to a solution with increasing amplitude in time. Figure 9.8 contains a comparison between the two methods. Note that the Forward Euler method uses 10 times as many time steps as the 4-th order Runge-Kutta method and is still much less accurate. A very much smaller time step is needed to limit the growth of the Forward Euler scheme for oscillating systems.




(a) (b)

**Fig. 9.8** Solution of an oscillating system ($u'' + u = 0$ formulated as system of two ODEs) by (a) the Forward Euler method with $\Delta t = 2\pi/200$; and (b) the 4-th order Runge-Kutta method with $\Delta t = 2\pi/20$.

### 9.4.9 Application 4: The Trajectory of a Ball

Exercise 1.14 derives the following two second-order differential equations for the motion of a ball (neglecting air resistance):

$$\frac{d^2x}{dt^2} = 0, \tag{9.35}$$

$$\frac{d^2y}{dt^2} = -g, \tag{9.36}$$

where $(x, y)$ is the position of the ball ($x$ is a horizontal measure and $y$ is a vertical measure), and $g$ is the acceleration of gravity. To use numerical methods for first-order equations, we must rewrite the system of two second-order equations as a system of four first-order equations. This is done by introducing to new unknowns, the velocities $v_x = dx/dt$ and $v_y = dy/dt$. We then have the first-order system of ODEs

$$\frac{dx}{dt} = v_x, \tag{9.37}$$

$$\frac{dv_x}{dt} = 0, \tag{9.38}$$

$$\frac{dy}{dt} = v_y, \tag{9.39}$$

$$\frac{dv_y}{dt} = -g\,. \tag{9.40}$$

The initial conditions are

$$x(0) = 0, \tag{9.41}$$

$$v_x(0) = v_0 \cos\theta, \tag{9.42}$$

$$y(0) = y_0, \tag{9.43}$$

$$v_y(0) = v_0 \sin\theta, \tag{9.44}$$

where $v_0$ is the initial magnitude of the velocity of the ball. The initial velocity has a direction that makes the angle $\theta$ with the horizontal.

The code related to this example is found in `app4_ball.py`. A function returning the right-hand side of our ODE system reads

```
def f(u, t):
    x, vx, y, vy = u
    g = 9.81
    return [vx, 0, vy, -g]
```

The main program for solving the ODEs can be set up as

```
v0 = 5
theta = 80*pi/180
u0 = [0, v0*cos(theta), 0, v0*sin(theta)]
T = 1.2
dt = 0.01
method = ForwardEuler(f, dt)
method.set_initial_condition(u0, 0)
u, t = method.solve(T)
```

Now, `u` is an array of 4-arrays `[x, vx, y, vy]`. Say we want to plot `x` as a function of time. We then have to extract all the `x` values as the first column in the two-dimensional `u` array:

```
x_values = u[:,0]
# or (slower):
x_values = array([x for x, vx, y, vy in u])
plot(t, x_values)
```

When a plot of the trajectory is desired, we need to plot the $y$ coordinates of the ball against the $x$ coordinates:

```
x_values = u[:,0]
y_values = u[:,2]
plot(x_values, y_values)
```

The exact solution is given by (1.5), so we can easily assess the accuracy of the numerical solution:

```
def exact(x):
    g = 9.81; y0 = u0[2]
    return x*tan(theta) - g*x**2/(2*v0**2)*1/(cos(theta))**2 + y0

plot(x_values, y_values, "r-",
     x_values, exact(x_values), "b-",
     legend=("numerical", "exact"),
     title="dt=%g" % dt)
```

Figure 9.9 shows a comparison of the numerical and the exact solution in this simple test problem. Note that even if we are just interested in $y$ as a function of $x$, we first need to solve the complete ODE system for the arrays x, vx, y, vy before we have x and y and can plot these.

The real strength of the numerical approach is the ease with which we can add air resistance and lift to the system of ODEs. Insight in physics is necessary to derive what the additional terms are, but implementing the terms is trivial in our test program above.

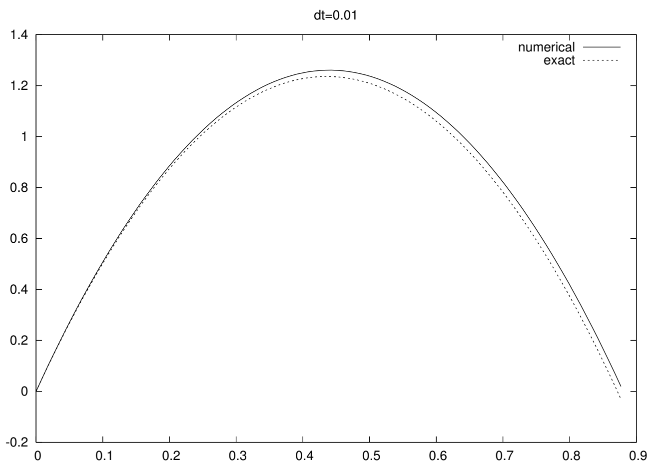



**Fig. 9.9** The trajectory of a ball solved as a system of four ODEs by the Forward Euler method.

## 9.5 Class Hierarchy for Geometric Shapes

Our next examples concern drawing geometric shapes. We know from Chapter 4 how to draw curves $y = f(x)$, but the point now is to construct some convenient software tools for drawing squares, circles, arcs, springs, wheels, and other shapes. With these tools we can create figures describing physical systems, for instance. Classes are very suitable for implementing the software because each shape is naturally associ-

ated with a class, and the various classes are related to each other through a natural hierarchy.

### 9.5.1 Using the Class Hierarchy

Before we dive into implementation details, let us first decide upon the interface we want to have for drawing various shapes. We start out by defining a rectangular area in which we will draw our figures. This is done by

```
from shapes import *
set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)
```

A line from $(0, 0)$ to $(1, 1)$ is defined by

```
l1 = Line(start=(0,0), stop=(1,1))  # define line
l1.draw()        # make plot data
display()        # display the plot data
```

A rectangle whose lower left corner is at $(0, 1)$, and where the width is 3 and the height is 5, is constructed by

```
r1 = Rectangle(lower_left_corner=(0,1), width=3, height=5)
r1.draw()
display()
```

A circle with center at $(5, 2)$ and unit radius, along with a wheel, is drawn by the code

```
Circle(center=(5,7), radius=1).draw()
Wheel(center=(6,2), radius=2, inner_radius=0.5, nlines=7).draw()
display()
hardcopy('tmp')   # create PNG file tmp.png
```

The latter line also makes a hardcopy of the figure in a PNG file. Figure 9.10 shows the resulting drawing after these commands.

We can change the color and thickness of the lines and also fill circles, rectangles, etc. with a color. Figure 9.11 shows the result of the following example, where we first define elements in the figure and then adjust the line color and other properties prior to calling the `draw` methods:

```
r1 = Rectangle(lower_left_corner=(0,1), width=3, height=5)
c1 = Circle(center=(5,7), radius=1)
w1 = Wheel(center=(6,2), radius=2, inner_radius=0.5, nlines=7)
c2 = Circle(center=(7,7), radius=1)
filled_curves(True)
c1.draw()              # filled red circle
set_linecolor('blue')
r1.draw()              # filled blue rectangle
set_linecolor('aqua')
c2.draw()              # filled aqua/cyan circle
# add thick aqua line around rectangle:
filled_curves(False)
```

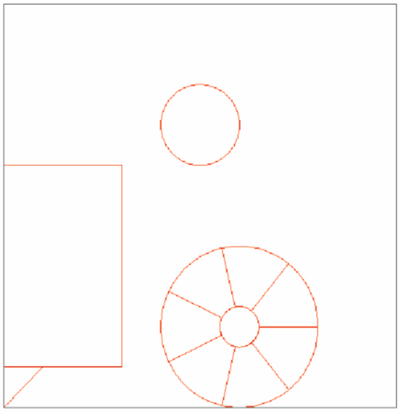


**Fig. 9.10** Result of a simple drawing session with shapes from the `Shape` class hierarchy.

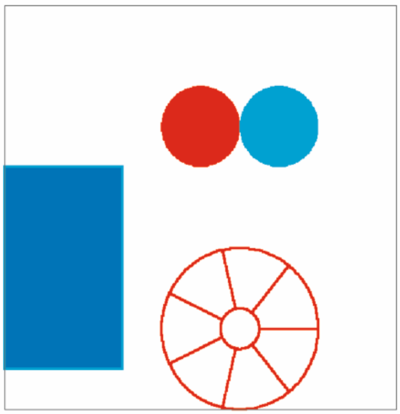


**Fig. 9.11** Redrawing of some shapes from Figure 9.10 with some thicker lines and different colors.

```
set_linewidth(4)
r1.draw()
set_linecolor('red')
w1.draw()
display()
```

### 9.5.2 Overall Design of the Class Hierarchy

Let us have a class `Shape` as superclass for all specialized shapes. Class `Line` is a subclass of `Shape` and represents the simplest shape: a stright line between two points. Class `Rectangle` is another subclass of `Shape`, implementing the functionality needed to specify the four lines of a rectangle. Class `Circle` can be yet another subclass of `Shape`, or we may have a class `Arc` and let `Circle` be a subclass of `Arc` since a circle is an arc of 360 degrees. Class `Wheel` is also subclass of `Shape`, but it

contains naturally two `Circle` instances for the inner and outer circles, plus a set of `Line` instances going from the inner to the outer circles.

The discussion in the previous paragraph shows that a subclass in the `Shape` hierarchy typically contains a list of other subclass instances, *or* the shape is a primitive, such as a line, circle, or rectangle, where the geometry is defined through a set of $(x, y)$ coordinates rather than through other `Shape` instances. It turns out that the implementation is simplest if we introduce a class `Curve` for holding a primitive shape defined by $(x, y)$ coordinates. Then all other subclasses of `Shape` can have a list `shapes` holding the various instances of subclasses of `Shape` needed to build up the geometric object. The `shapes` attribute in class `Circle` will contain one `Curve` instance for holding the coordinates along the circle, while the `shapes` attribute in class `Wheel` contains two `Circle` instances and a number of `Line` instances. Figures 9.12 and 9.13 display two UML drawings of the `shapes` class hierarchy where we can get a view of how `Rectangle` and `Wheel` relate to other classes: the darkest arrows represent is-a relationship while the lighter arrows represent has-a relationship.

All instances in the `Shape` hierarchy must have a `draw` method. The `draw` method in class `Curve` plots the $(x, y)$ coordinates as a curve, while the `draw` method in all other classes simply do a

```
for shape in self.shapes:
    shape.draw()
```

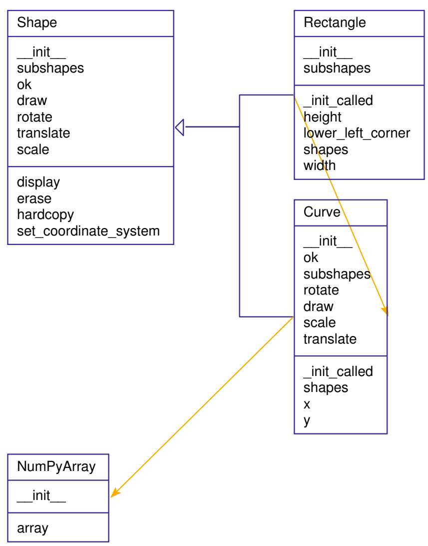



**Fig. 9.12** UML diagram of parts of the `shapes` hierarchy. Classes `Rectangle` and `Curve` are subclasses of `Shape`. The darkest arrow with the biggest arrowhead indicates inheritance and is-a relationship: `Rectangle` and `Curve` are both also `Shape`. The lighter arrow indicates has-a relationship: `Rectangle` has a `Curve`, and a `Curve` has a `NumPyArray`.

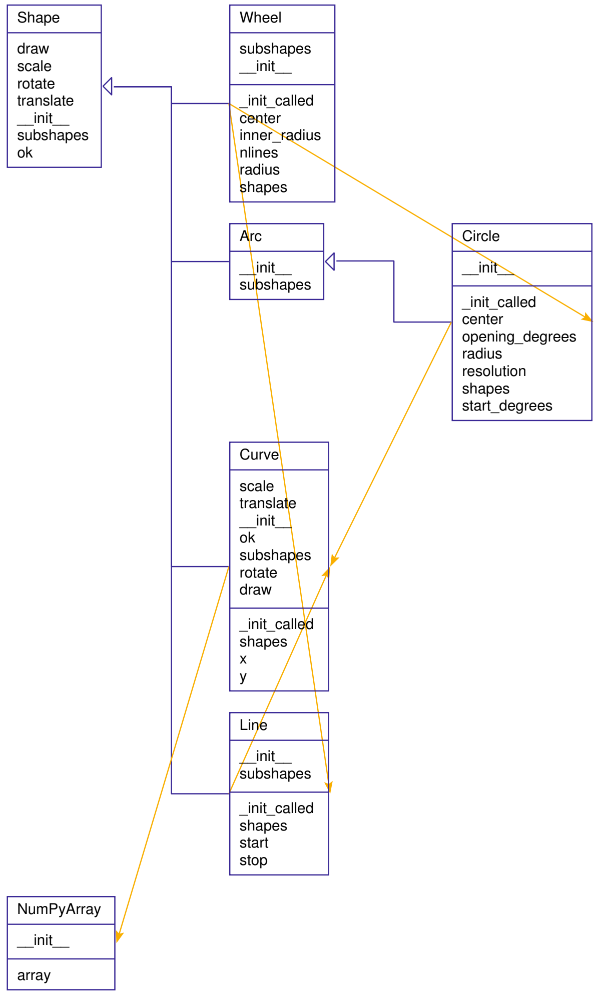



**Fig. 9.13** This is a variant of Figure 9.12 where we display how class `Wheel` relates to other classes in the `shapes` hierarchy. `Wheel` is a `Shape`, like `Arc`, `Line`, and `Curve`, but `Wheel` contains `Circle` and `Line` objects, while the `Circle` and `Line` objects have a `Curve`, which has a `NumPyArray`. We also see that `Circle` is a subclass of `Arc`.

### 9.5.3 The Drawing Tool

We have in Chapter 4 introduced the Easyviz tool for plotting graphs. This tool is quite well suited for drawing geometric shapes defined in terms of curves, but when drawing shapes we often want to skip ticmarks on the axis, labeling of the curves and axis, and perform other adjustments. Instead of using Easyviz, which aims at function plotting, we have decided to use a plotting tool directly and fine-tune the few commands we need for drawing shapes.

A simple plotting tool for shapes is based on Gnuplot and implemented in class `GnuplotDraw` in the file `GnuplotDraw.py`. This class has the following user interface:

```
class GnuplotDraw:
    def __init__(self, xmin, xmax, ymin, ymax):
        """Define the drawing area [xmin,xmax]x[ymin,ymax]."""

    def define_curve(self, x, y):
        """Define a curve with coordinates x and y (arrays)."""

    def erase(self):
        """Erase the current figure."""

    def display(self):
        """Display the figure."""

    def hardcopy(self, name):
        """Save figure in PNG file name.png."""

    def set_linecolor(self, color):
        """Change the color of lines."""

    def set_linewidth(self, width):
        """Change the line width (int, starts at 1)."""

    def filled_curves(self, on=True):
        """Fill area inside curves with current line color."""
```

One can easily make a similar class with an identical interface that applies another plotting package than Gnuplot to create the drawings. In particular, encapsulating the drawing actions in such a class makes it trivial to change the drawing program in the future. The program pieces that apply a drawing tool like `GnuplotDraw` remain the same. This is an important strategy to follow, especially when developing larger software systems.

### 9.5.4 Implementation of Shape Classes

Our superclass `Shape` can naturally hold a coordinate system specification, i.e., the rectangle in which other shapes can be drawn. This area is fixed for all shapes, so the associated variables should be static and the method for setting them should also be static (see Chapter 7.7 for static attributes and methods). It is also natural that class `Shape` holds access to a drawing tool, in our case a `GnuplotDraw` instance. This object is also static. However, it can be an advantage to mirror the static attributes and methods as global variables and functions in the `shapes` modules. Users not familiar with static class items can drop the `Shape` prefix and just use plain module variables and functions. This is what we do in the application examples.

Class `Shape` defines an imporant method, `draw`, which just calls the `draw` method for all subshapes that build up the current shape.

Here is a brief view of class `Shape`[9]:

[9] We have for simplicity omitted the static attributes and methods. These can be viewed in the `shapes.py` file.

```
class Shape:
    def __init__(self):
        self.shapes = self.subshapes()
        if isinstance(self.shapes, Shape):
            self.shapes = [self.shapes]  # turn to list

    def subshapes(self):
        """Define self.shapes as list of Shape instances."""
        raise NotImplementedError(self.__class__.__name__)

    def draw(self):
        for shape in self.shapes:
            shape.draw()
```

In class `Shape` we require the `shapes` attribute to be a list, but if the `subshape` method in subclasses returns just one instance, this is automatically wrapped in a list in the constructor.

First we implement the special case class `Curve`, which does not have subshapes but instead $(x, y)$ coordinates for a curve:

```
class Curve(Shape):
    """General (x,y) curve with coordintes."""
    def __init__(self, x, y):
        self.x, self.y = x, y
        # turn to Numerical Python arrays:
        self.x = asarray(self.x, float)
        self.y = asarray(self.y, float)
        Shape.__init__(self)

    def subshapes(self):
        pass # geometry defined in constructor
```

The simplest ordinary `Shape` class is `Line`:

```
class Line(Shape):
    def __init__(self, start, stop):
        self.start, self.stop = start, stop
        Shape.__init__(self)

    def subshapes(self):
        x = [self.start[0], self.stop[0]]
        y = [self.start[1], self.stop[1]]
        return Curve(x,y)
```

The code in this class works with `start` and `stop` as tuples, lists, or arrays of length two, holding the end points of the line. The underlying `Curve` object needs only these two end points.

A rectangle is represented by a slightly more complicated class, having the lower left corner, the width, and the height of the rectangle as attributes:

```
class Rectangle(Shape):
    def __init__(self, lower_left_corner, width, height):
        self.lower_left_corner = lower_left_corner  # 2-tuple
        self.width, self.height = width, height
        Shape.__init__(self)

    def subshapes(self):
        ll = self.lower_left_corner  # short form
```

```
        x = [ll[0], ll[0]+self.width,
             ll[0]+self.width, ll[0], ll[0]]
        y = [ll[1], ll[1], ll[1]+self.height,
             ll[1]+self.height, ll[1]]
        return Curve(x,y)
```

Class `Circle` needs many coordinates in its `Curve` object in order to display a smooth circle. We can provide the number of straight line segments along the circle as a parameter `resolution`. Using a default value of 180 means that each straight line segment approximates an arc of 2 degrees. This resolution should be sufficient for visual purposes. The set of coordinates along a circle with radius $R$ and center $(x_0, y_0)$ is defined by

$$x = x_0 + R\cos(t), \tag{9.45}$$

$$y = y_0 + R\sin(t), \tag{9.46}$$

for `resolution+1` $t$ values between 0 and $2\pi$. The vectorized NumPy code for computing the coordinates becomes

```
t = linspace(0, 2*pi, self.resolution+1)
x = x0 + R*cos(t)
y = y0 + R*sin(t)
```

The complete `Circle` class is shown below:

```
class Circle(Shape):
    def __init__(self, center, radius, resolution=180):
        self.center, self.radius = center, radius
        self.resolution = resolution
        Shape.__init__(self)

    def subshapes(self):
        t = linspace(0, 2*pi, self.resolution+1)
        x0 = self.center[0];  y0 = self.center[1]
        R = self.radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        return Curve(x,y)
```

We can also introduce class `Arc` for drawing the arc of a circle. Class `Arc` could be a subclass of `Circle`, extending the latter with two additional parameters: the opening of the arc (in degrees) and the starting $t$ value in (9.45)–(9.46). The implementation of class `Arc` will then be almost a copy of the implementation of class `Circle`. The `subshapes` method will just define a different `t` array.

Another view is to let class `Arc` be a subclass of `Shape`, and `Circle` a subclass of `Arc`, since a circle is an arc of 360 degrees. Let us employ this idea:

```
class Arc(Shape):
    def __init__(self, center, radius,
                 start_degrees, opening_degrees, resolution=180):
        self.center = center
        self.radius = radius
```

```
        self.start_degrees = start_degrees*pi/180
        self.opening_degrees = opening_degrees*pi/180
        self.resolution = resolution
        Shape.__init__(self)

    def subshapes(self):
        t = linspace(self.start_degrees,
                     self.start_degrees + self.opening_degrees,
                     self.resolution+1)
        x0 = self.center[0];  y0 = self.center[1]
        R = self.radius
        x = x0 + R*cos(t)
        y = y0 + R*sin(t)
        return Curve(x,y)

class Circle(Arc):
    def __init__(self, center, radius, resolution=180):
        Arc.__init__(self, center, radius, 0, 360, resolution)
```

In this latter implementation, we save a lot of code in class `Circle` since all of class `Arc` can be reused.

Class `Wheel` may conceptually be a subclass of `Circle`. One circle, say the outer, is inherited and the subclass must have the inner circle as an attribute. Because of this "asymmetric" representation of the two circles in a wheel, we find it more natural to derive `Wheel` directly from `Shape`, and have the two circles as two attributes of type `Circle`:

```
class Wheel(Shape):
    def __init__(self, center, radius, inner_radius=None, nlines=10):
        self.center = center
        self.radius = radius
        if inner_radius is None:
            self.inner_radius = radius/5.0
        else:
            self.inner_radius = inner_radius
        self.nlines = nlines
        Shape.__init__(self)
```

If the radius of the inner circle is not defined (`None`) we take it as 1/5 of the radius of the outer circle. The wheel is naturally composed of two `Circle` instances and `nlines` `Line` instances:

```
    def subshapes(self):
        outer = Circle(self.center, self.radius)
        inner = Circle(self.center, self.inner_radius)
        lines = []
        t = linspace(0, 2*pi, self.nlines)
        Ri = self.inner_radius;  Ro = self.radius
        x0 = self.center[0];  y0 = self.center[1]
        xinner = x0 + Ri*cos(t)
        yinner = y0 + Ri*sin(t)
        xouter = x0 + Ro*cos(t)
        youter = y0 + Ro*sin(t)
        lines = [Line((xi,yi),(xo,yo)) for xi, yi, xo, yo in \
                  zip(xinner, yinner, xouter, youter)]
        return [outer, inner] + lines
```

For the fun of it, we can implement other shapes, say a sine wave

$$y = m + A\sin kx, \quad k = 2\pi/\lambda,$$

where $\lambda$ is the wavelength of the sine waves, $A$ is the wave amplitude, and $m$ is the mean value of the wave. The class looks like

```
class Wave(Shape):
    def __init__(self, xstart, xstop,
                 wavelength, amplitude, mean_level):
        self.xstart = xstart
        self.xstop = xstop
        self.wavelength = wavelength
        self.amplitude = amplitude
        self.mean_level = mean_level
        Shape.__init__(self)

    def subshapes(self):
        npoints = (self.xstop - self.xstart)/(self.wavelength/61.0)
        x = linspace(self.xstart, self.xstop, npoints)
        k = 2*pi/self.wavelength # frequency
        y = self.mean_level + self.amplitude*sin(k*x)
        return Curve(x,y)
```

With this and the previous example, you should be in a position to write your own subclasses. Exercises 9.35–9.39 suggest some smaller projects.

*Functions for Controlling Lines, Colors, etc.* The `shapes` module containing class `Shape` and all subclasses mentioned above, also offers some additional functions that do not depend on any particular shape:

- `display()` for displaying the defined figures so far (all figures whose `draw` method is called).
- `erase()` for ereasing the current figure.
- `hardcopy(name)` for saving the current figure to a PNG file `name.png`.
- `set_linecolor(color)` for setting the color of lines, where `color` is a string like `'red'` (default), `'blue'`, `'green'`, `'aqua'`, `'purple'`, `'yellow'`, and `'black'`.
- `set_linewidth(width)` for setting the width of a line, measured as an integer (default is 2).
- `filled_curves(on)` for turrning on (`on=True`) or off (`on=False`) whether the area inside a shape should be filled with the current line color.

Actually, the functions above are static methods in class `Shape` (cf. Chapter 7.7), and they are just mirrored as global functions[10] in the `shapes` module. Users without knowledge of static methods do not need to use the `Shape` prefix for reaching this functionality.

### 9.5.5 Scaling, Translating, and Rotating a Figure

The real power of object-oriented programming will be obvious in a minute when we, with a few lines of code, suddenly can equip *all* shape

[10] You can look into `shapes.py` to see how we automate the duplication of static methods as global functions.

objects with additional functionality for scaling, translating, and rotating the figure.

*Scaling.* Let us first treat the simplest of the three cases: scaling. For a `Curve` instance containing a set of $n$ coordinates $(x_i, y_i)$ that make up a curve, scaling by a factor $a$ means that we multiply all the $x$ and $y$ coordinates by $a$:

$$x_i \leftarrow ax_i, \quad y_i \leftarrow ay_i, \quad i = 0, \ldots, n-1\,.$$

Here we apply the arrow as an assignment operator. The corresponding Python implementation in class `Curve` reads

```
class Curve:
    ...
    def scale(self, factor):
        self.x = factor*self.x
        self.y = factor*self.y
```

Note here that `self.x` and `self.y` are Numerical Python arrays, so that multiplication by a scalar number `factor` is a vectorized operation.

In an instance of a subclass of `Shape`, the meaning of a method `scale` is to run through all objects in the list `self.shapes` and ask each object to scale itself. This is the same delegation of actions to subclass instances as we do in the `draw` method, and all objects, except `Curve` instances, can share the same implementation of the `scale` method. Therefore, we place the `scale` method in the superclass `Shape` such that all subclasses can inherit this method. Since `scale` and `draw` are so similar, we can easily implement the `scale` method in class `Shape` by copying and editing the `draw` method:

```
class Shape:
    ...
    def scale(self, factor):
        for shape in self.shapes:
            shape.scale(factor)
```

This is all we have to do in order to equip all subclasses of `Shape` with scaling functionality! But why is it so easy? All subclasses inherit `scale` from class `Shape`. Say we have a subclass instance `s` and that we call `s.scale(factor)`. This leads to calling the inherited `scale` method shown above, and in the `for` loop we call the `scale` method for each `shape` object in the `self.shapes` list. If `shape` is not a `Curve` object, this procedure repeats, until we hit a `shape` that is a `Curve`, and then the scaling on that set of coordinates is performed.

*Translation.* A set of coordinates $(x_i, y_i)$ can be translated $x$ units in the $x$ direction and $y$ units in the $y$ direction using the formulas

$$x_i \leftarrow x + x_i, \quad y_i \leftarrow y + y_i, \quad i = 0, \ldots, n-1\,.$$

The corresponding Python implementation in class `Curve` becomes

```
class Curve:
    ...
    def translate(self, x, y):
        self.x = x + self.x
        self.y = y + self.y
```

The translation operation for a shape object is very similar to the scaling and drawing operations. This means that we can implement a common method `translate` in the superclass `Shape`. The code is parallel to the `scale` method:

```
class Shape:
    ....
    def translate(self, x, y):
        for shape in self.shapes:
            shape.translate(x, y)
```

*Rotation.* Rotating a figure is more complicated than scaling and translating. A counter clockwise rotation of $\theta$ degrees for a set of coordinates $(x_i, y_i)$ is given by

$$\bar{x}_i \leftarrow x_i \cos\theta - y_i \sin\theta,$$
$$\bar{y}_i \leftarrow x_i \sin\theta + y_i \cos\theta\,.$$

This rotation is performed around the origin. If we want the figure to be rotated with respect to a general point $(x, y)$, we need to extend the formulas above:

$$\bar{x}_i \leftarrow x + (x_i - x)\cos\theta - (y_i - y)\sin\theta,$$
$$\bar{y}_i \leftarrow y + (x_i - x)\sin\theta + (y_i - y)\cos\theta\,.$$

The Python implementation in class `Curve`, assuming that $\theta$ is given in degrees and not in radians, becomes

```
    def rotate(self, angle, x=0, y=0):
        angle = angle*pi/180
        c = cos(angle);  s = sin(angle)
        xnew = x + (self.x - x)*c - (self.y - y)*s
        ynew = y + (self.x - x)*s + (self.y - y)*c
        self.x = xnew
        self.y = ynew
```

The `rotate` method in class `Shape` is identical to the `draw`, `scale`, and `translate` methods except that we have other arguments:

```
class Shape:
    ....
    def rotate(self, angle, x=0, y=0):
        for shape in self.shapes:
            shape.rotate(angle, x, y)
```

*Application: Rolling Wheel.* To demonstrate the effect of translation and rotation we can roll a wheel on the screen. First we draw the wheel and rotate it a bit to demonstrate the basic operations:

```
center = (6,2)   # the wheel's center point
w1 = Wheel(center=center, radius=2, inner_radius=0.5, nlines=7)
# rorate the wheel 2 degrees around its center point:
w1.rotate(angle=2, center[0], center[1])
w1.draw()
display()
```

Now we want to roll the wheel by making many such small rotations. At the same time we need to translate the wheel since rolling an arc length $L = R\theta$, where $\theta$ is the rotation angle (in radians) and $R$ is the outer radius of the wheel, implies that the center point moves a distance $L$ to the left ($\theta > 0$ means counter clockwise rotation). In code we must therefore combine rotation with translation:

```
L = radius*angle*pi/180  # translation = arc length
w1.rotate(angle, center[0], center[1])
w1.translate(-L, 0)
center = (center[0] - L, center[1])
```

We are now in a position to put the rotation and translation operations in a `for` loop and make a complete function:

```
def rolling_wheel(total_rotation_angle):
    """Animation of a rotating wheel."""
    set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)

    center = (6,2)
    radius = 2.0
    angle = 2.0
    w1 = Wheel(center=center, radius=radius,
               inner_radius=0.5, nlines=7)
    for i in range(int(total_rotation_angle/angle)):
        w1.draw()
        display()

        L = radius*angle*pi/180  # translation = arc length
        w1.rotate(angle, center[0], center[1])
        w1.translate(-L, 0)
        center = (center[0] - L, center[1])

        erase()
```

To control the visual "velocity" of the wheel, we can insert a pause between each frame in the `for` loop. A call to `time.sleep(s)`, where `s` is the length of the pause in seconds, can do this for us.

Another convenient feature is to save each frame drawn in the `for` loop as a hardcopy in PNG format and then, after the loop, make an animated GIF file based on the individual PNG frames. The latter operation is performed either by the `movie` function from `scitools.std` or by the `convert` program from the ImageMagick suite. With the latter you write the following command in a terminal window:

```
convert -delay 50 -loop 1000 xxx tmp_movie.gif
```

Here, `xxx` is a space-separated list of all the PNG files, and `tmp_movie.gif` is the name of the resulting animated GIF file. We can easily make `xxx` by collecting the names of the PNG files from the loop in a list object, and then join the names. The `convert` command can be run as an `os.system` call.

The complete `rolling_wheel` function, incorporating the mentioned movie making, will then be

```
def rolling_wheel(total_rotation_angle):
    """Animation of a rotating wheel."""
    set_coordinate_system(xmin=0, xmax=10, ymin=0, ymax=10)

    import time
    center = (6,2)
    radius = 2.0
    angle = 2.0
    pngfiles = []
    w1 = Wheel(center=center, radius=radius,
               inner_radius=0.5, nlines=7)
    for i in range(int(total_rotation_angle/angle)):
        w1.draw()
        display()

        filename = 'tmp_%03d' % i
        pngfiles.append(filename + '.png')
        hardcopy(filename)
        time.sleep(0.3)  # pause 0.3 sec

        L = radius*angle*pi/180  # translation = arc length
        w1.rotate(angle, center[0], center[1])
        w1.translate(-L, 0)
        center = (center[0] - L, center[1])

        erase()  # erase the screen before new figure

    cmd = 'convert -delay 50 -loop 1000 %s tmp_movie.gif' \
          % (' '.join(pngfiles))
    import commands
    failure, output = commands.getstatusoutput(cmd)
    if failure:  print 'Could not run', cmd
```

The last two lines run a command, from Python, as we would run the command in a terminal window. The resulting animated GIF file can be viewed with `animate tmp_movie.gif` as a command in a terminal window.

## 9.6 Summary

### 9.6.1 Chapter Topics

A subclass inherits everyting from its superclass, both attributes and methods. The subclass can add new attributes, overload methods, and thereby enrich or restrict functionality of the superclass.

*Subclass Example.* Consider class `Gravity` from Chapter 7.8.1 for representing the gravity force $GMm/r^2$ between two masses $m$ and $M$

being a distance $r$ apart. Suppose we want to make a class for the electric force between two charges $q_1$ and $q_2$, being a distance $r$ apart in a medium with permittivity $\epsilon_0$ is $Gq_1q_2/r^2$, where $G^{-1} = 4\pi\epsilon_0$. We use the approximate value $G = 8.99 \cdot 10^9$ Nm$^2$/C$^2$ (C is the Coulumb unit used to measure electric charges such as $q_1$ and $q_2$). Since the electric force is similar to the gravity force, we can easily implement the eletric force as a subclass of `Gravity`. The implementation just needs to redefine the value of $G$!

```
class CoulumbsLaw(Gravity):
    def __init__(self, q1, q2):
        Gravity.__init__(self, q1, q2)
        self.G = 8.99E9
```

We can now call the inherited `force(r)` method to compute the eletric force and the `visualize` method to make a plot of the force:

```
c = CoulumbsLaw(1E-6, -2E-6)
print 'Electric force:', c.force(0.1)
c.visualize(0.01, 0.2)
```

However, the `plot` method inherited from class `Gravity` has an inappropriate title referring to "Gravity force" and the masses $m$ and $M$. An easy fix could be to have the plot title as an attribute set in the constructor. The subclass can then override the contents of this attribute, as it overrides `self.G`. It is quite common to discover that a class needs adjustments if it is to be used as superclass.

*Subclassing in General.* The typical sketch of creating a subclass goes as follows:

```
class SuperClass:
    def __init__(self, p, q):
        self.p, self.q = p, q

    def where(self):
        print 'In superclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return self.p*x + self.q


class SubClass(SuperClass):
    def __init__(self, p, q, a):
        SuperClass.__init__(self, p, q)
        self.a = a

    def where(self):
        print 'In subclass', self.__class__.__name__

    def compute(self, x):
        self.where()
        return SuperClass.compute(self, x) + self.a*x**2
```

This example shows how a subclass extends a superclass with one attribute (`a`). The subclass' `compute` method calls the corresponding su-

perclass method, as well as the overloaded method `where`. Let us invoke the `compute` method through superclass and subclass instances:

```
>>> super = SuperClass(1, 2)
>>> sub = SubClass(1, 2, 3)
>>> v1 = super.compute(0)
In superclass SuperClass
>>> v2 = sub.compute(0)
In subclass SubClass
In subclass SubClass
```

Observe that in the subclass `sub`, method `compute` calls `self.where`, which translates to the `where` method in `SubClass`. Then the `compute` method in `SuperClass` is invoked, and this method also makes a `self.where` call, which is a call to `SubClass`' `where` method (think of what `self` is here, it is `sub`, so it is natural that we get `where` in the subclass (`sub.where`) and not `where` in the superclass part of `sub`).

In this example, classes `SuperClass` and `SubClass` constitute a class hierarchy. Class `SubClass` inherits the attributes `p` and `q` from its superclass, and overrides the methods `where` and `compute`.

### 9.6.2 Summarizing Example: Input Data Reader

The summarizing example of this chapter concerns a class hierarchy for simplifying reading input data into programs. Input data may come from several different sources: the command line, a file, or from a dialog with the user, either of `input` form or in a graphical user interface (GUI). Therefore it makes sense to create a class hierarchy where subclasses are specialized to read from different sources and where the common code is placed in a superclass. The resulting tool will make it easy for you to let your programs read from many different input sources by adding just a few lines.

*Problem.* Let us motive the problem by a case where we want to write a program for dumping $n$ function values of $f(x)$ to a file for $x \in [a, n]$. The core part of the program typically reads

```
outfile = open(filename, 'w')
from numpy import linspace
for x in linspace(a, b, n):
    outfile.write('%12g  %12g\n' % (x, f(x)))
outfile.close()
```

Our purpose is to read data into the variables `a`, `b`, `n`, `filename`, and `f`. For the latter we want to specify a formula and use the `StringFunction` tool (Chapter 3.1.4) to make the function `f`:

```
from scitools.StringFunction import StringFunction
f = StringFunction(formula)
```

How can we read `a`, `b`, `n`, `formula`, and `filename` conveniently into the program?

The basic idea is that we place the input data in a dictionary, and create a tool that can update this dictionary from sources like the command line, a file, a GUI, etc. Our dictionary is then

```
p = dict(formula='x+1', a=0, b=1, n=2, filename='tmp.dat')
```

This dictionary specifies the names of the input parameters to the program and the default values of these parameters.

Using the tool is a matter of feeding `p` into the constructor of a subclass in the tools' class hierarchy and extract the parameters into, for example, distinct variables:

```
inp = Subclassname(p)
a, b, filename, formula, n = inp.get_all()
```

Depending on what we write as `Subclassname`, the five variables can be read from the command line, the terminal window, a file, or a GUI. The task now is to implement a class hierarchy to facilitate the described flexible reading of input data.

*Solution.* We first create a very simple superclass `ReadInput`. Its main purpose is to store the parameter dictionary as an attribute, provide a method `get` to extract single values, and a method `get_all` to extract all parameters into distinct variables:

```
class ReadInput:
    def __init__(self, parameters):
        self.p = parameters

    def get(self, parameter_name):
        return self.p[parameter_name]

    def get_all(self):
        return [self.p[name] for name in sorted(self.p)]

    def __str__(self):
        import pprint
        return pprint.pformat(self.p)
```

Note that we in the `get_all` method must sort the keys in `self.p` such that the list of returned variables is well defined. In the calling program we can then list variables in the same order as the alphabetic order of the parameter names, for example:

```
a, b, filename, formula, n = inp.get_all()
```

The `__str__` method applies the `pprint` module to get a pretty print of all the parameter names and their values.

Class `ReadInput` cannot read from any source – subclasses are supposed to do this. The forthcoming text describes various types of subclasses for various types of reading input.

*Prompting the User.* The perhaps simplest way of getting data into a program is to use `raw_input`. We then prompt the user with a text `Give name:` and get an appropriate object back (recall that strings must be enclosed in quotes). The subclass `PromptUser` for doing this then reads

```
class PromptUser(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._prompt_user()

    def _prompt_user(self):
        for name in self.p:
            self.p[name] = eval(raw_input("Give " + name + ": "))
```

Note the underscore in `_prompt_user`: the underscore signifies that this is a "private" method in the `PromptUser` class, not intended to be called by users of the class.

There is a major difficulty with using `eval` on the input from the user. When the input is intended to be a string object, such as a filename, say `tmp.inp`, the program will perform the operation `eval(tmp.inp)`, which leads to an exception because `tmp.inp` is treated as a variable `inp` in a module `tmp` and not as the string `'tmp.inp'`. To solve this problem, we use the `str2obj` function from the `scitools.misc` module. This function will return the right Python object also in the case where the argument should result in a string object (see Chapter 3.6.1 for some information about `str2obj`). The bottom line is that `str2obj` acts as a safer `eval(raw_input(...))` call. The key assignment in class `PromptUser` is then changed to

```
self.p[name] = str2obj(raw_input("Give " + name + ": "))
```

*Reading from File.* We can also place `name = value` commands in a file and load this information into the dictionary `self.p`. An example of a file can be

```
formula    = sin(x) + cos(x)
filename   = tmp.dat
a          = 0
b          = 1
```

In this example we have omitted `n`, so we rely on its default value.

A problem is how to give the filename. The easy way out of this problem is to read from standard input, and just redirect standard input from a file when we run the program. For example, if the filename is `tmp.inp`, we run the program as follows in a terminal window[11]

Terminal

```
Unix/DOS> python myprog.py < tmp.inp
```

[11] The redirection of standard input from a file does not work in IPython so we are in this case forced to run the program in a terminal window.

To interpret the contents of the file, we read line by line, split each line with respect to `=`, use the left-hand side as the parameter name and the right-hand side as the corresponding value. It is important to strip away unnecessary blanks in the name and value. The complete class now reads

```
class ReadInputFile(ReadInput):
    def __init__(self, parameters):
        ReadInput.__init__(self, parameters)
        self._read_file()

    def _read_file(self, infile=sys.stdin):
        for line in infile:
            if "=" in line:
                name, value = line.split("=")
                self.p[name.strip()] = str2obj(value.strip())
```

A nice feature with reading from standard input is that if we do not redirect standard input to a file, the program will prompt the user in the terminal window, where the user can give commands of the type `name = value` for setting selected input data. A `Ctrl-D` is needed to terminate the interactive session in the terminal window and continue execution of the program.

*Reading from the Command Line.* For input from the command line we assume that parameters and values are given as option-value pairs, e.g., as in

```
--a 1 --b 10 --n 101 --formula "sin(x) + cos(x)"
```

We apply the `getopt` module (Chapter 3.2.4) to parse the command-line arguments. The list of legal option names must be constructed from the list of keys in the `self.p` dictionary. The complete class takes the form

```
class ReadCommandLine(ReadInput):
    def __init__(self, parameters):
        self.sys_argv = sys.argv[1:]  # copy
        ReadInput.__init__(self, parameters)
        self._read_command_line()

    def _read_command_line(self):
        # make getopt list of options:
        option_names = [name + "=" for name in self.p]
        try:
            options, args = getopt.getopt(self.sys_argv,
                                          '', option_names)
        except getopt.GetoptError, e:
            print 'Error in command-line option:\n', e
            sys.exit(1)

        for option, value in options:
            for name in self.p:
                if option == "--" + name:
                    self.p[name] = str2obj(value)
```

*Reading from a GUI.* We can with a little extra effort also make a graphical user interface (GUI) for reading the input data. An example of a user interface is displayed in Figure 9.14. Since the technicalities of the implementation is beyond the scope of this book, we do not show the subclass `GUI` that creates the GUI and loads the user input into the `self.p` dictionary.

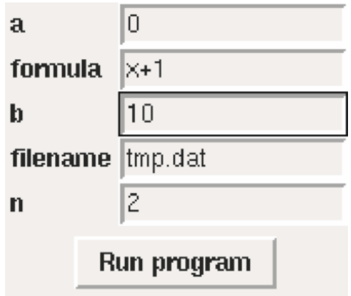



**Fig. 9.14** Screen dump of a graphical user interface to read input data into a program (class `GUI` in the `ReadInput` hierarchy).

*More Flexibility in the Superclass.* Some extra flexibility can easily be added to the `get` method in the superclass. Say we want to extract a variable number of parameters:

```
a, b, n = inp.get('a', 'b', 'n')  # 3 variables
n = inp.get('n')                  # 1 variable
```

The key to this extension is to use a variable number of arguments as explained in Appendix E.5.1:

```
class ReadInput:
    ...
    def get(self, *parameter_names):
        if len(parameter_names) == 1:
            return self.p[parameter_names[0]]
        else:
            return [self.p[name] for name in parameter_names]
```

*Demonstrating the Tool.* Let us show how we can use the classes in the `ReadInput` hiearchy. We apply the motivating example described earlier. The name of the program is `demo_ReadInput.py`. As first command-line argument it takes the name of the input source, given as the name of a subclass in the `ReadInput` hierarchy. The code for loading input data from any of the sources supported by the `ReadInput` hierarchy goes as follows:

```
p = dict(formula='x+1', a=0, b=1, n=2, filename='tmp.dat')
from ReadInput import *
input_reader = eval(sys.argv[1])  # PromptUser, ReadInputFile, ...
del sys.argv[1]  # otherwise getopt does not work properly...
```

```
inp = input_reader(p)
a, b, filename, formula, n = inp.get_all()
print inp
```

Note how convenient `eval` is to automatically create the right subclass for reading input data.

Our first try on running this program applies the `PromptUser` class:

Terminal

```
demo_ReadInput.py PromptUser
Give a: 0
Give formula: sin(x) + cos(x)
Give b: 10
Give filename: function_data
Give n: 101
{'a': 0,
 'b': 10,
 'filename': 'function_data',
 'formula': 'sin(x) + cos(x)',
 'n': 101}
```

The next example reads data from a file `tmp.inp` with the same contents as shown under the *Reading from File* paragraph above[12].

Terminal

```
demo_ReadInput.py ReadFileInput < tmp.inp
{'a': 0, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

We can also drop the redirection of standard input to a file, and instead run an interactive session in IPython or the terminal window:

Terminal

```
demo_ReadInput.py ReadFileInput
n = 101
filename = myfunction_data_file.dat
^D
{'a': 0,
 'b': 1,
 'filename': 'myfunction_data_file.dat',
 'formula': 'x+1',
 'n': 101}
```

Note that `Ctrl-D` is needed to end the interactive session with the user and continue program execution.

Command-line arguments can also be specified:

Terminal

```
demo_ReadInput.py ReadCommandLine \
          --a -1 --b 1 --formula "sin(x) + cos(x)"
{'a': -1, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

Finally, we can run the program with a GUI,

[12] This command with redirection from file must be run from a standard terminal window, not in an interactive IPython session.

Terminal

```
demo_ReadInput.py GUI
{'a': -1, 'b': 10, 'filename': 'tmp.dat',
 'formula': 'x+1', 'n': 2}
```

The GUI is shown in Figure 9.14.

Fortunately, it is now quite obvious how to apply the `ReadInput` hierarchy of classes in your own programs to simplify input. Especially in applications with a large number of parameters one can initially define these in a dictionary and then automatically create quite comprehensive user interfaces where the user can specify only some subset of the parameters (if the default values for the rest of the parameters are suitable).

## 9.7 Exercises

**Exercise 9.1.** *Demonstrate the magic of inheritance.*

Consider class `Line` from Chapter 9.1.1 and a subclass `Parabola0` defined as

```
class Parabola0(Line):
    pass
```

That is, class `Parabola0` does not have any own code, but it inherits from class `Line`. Demonstrate in a program or interactive session, using methods from Chapter 7.6.5, that an instance of class `Parabola0` contains everything (i.e., all attributes and methods) that an instance of class `Line` contains. Name of program file: `dir_subclass.py`. ◇

**Exercise 9.2.** *Inherit from classes in Ch. 9.1.*

The task in this exercise is to make a class `Cubic` for cubic functions

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

with a call operator and a `table` method as in classes `Line` and `Parabola` from Chapter 9.1. Implement class `Cubic` by inheriting from class `Parabola`, and call up functionality in class `Parabola` in the same way as class `Parabola` calls up functionality in class `Line`.

Make a similar class `Poly4` for 4-th degree polynomials

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

by inheriting from class `Cubic`. Insert `print` statements in all the `__call__` methods where you print out in which class you are. Evaluate cubic and a 4-th degree polynomial at a point, and observe the printouts from all the superclasses. Name of program file: `Cubic_Poly4.py`.
◇

**Exercise 9.3.** *Inherit more from classes in Ch. 9.1.*

Implement a class for the function $f(x) = A\sin(wx) + ax^2 + bx + c$. The class should have a call operator for evaluating the function for some argument $x$, and a constructor that takes the function parameters `A`, `w`, `a`, `b`, and `c` as arguments. Also a `table` method as in classes `Line` and `Parabola` should be present. Implement the class by deriving it from class `Parabola` and call up functionality already implemented in class `Parabola` whenever possible. Name of program file: `sin_plus_quadratic.py`. ◇

**Exercise 9.4.** *Reverse the class hierarchy from Ch. 9.1.*

Let class `Polynomial` from Chapter 7.3.7 be a superclass and implement class `Parabola` as a subclass. The constructor in class `Parabola` should take the three coefficients in the parabola as separate arguments. Try to reuse as much code as possible from the superclass in the subclass. Implement class `Line` as a subclass specialization of class `Parabola` (let the constructor take the two coefficients as two separate arguments).

Which class design do you prefer – class `Line` as a subclass of `Parabola` and `Polynomial`, or `Line` as a superclass with extensions in subclasses? Name of program file: `Polynomial_hier.py`. ◇

**Exercise 9.5.** *Super- and subclass for a point.*

A point $(x, y)$ in the plane can be represented by a class:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

We can extend the `Point` class to also contain the representation of the point in polar coordinates. To this end, create a subclass `PolarPoint` whose constructor takes the polar representation of a point, $(r, \theta)$, as arguments. Store $r$ and $\theta$ as attributes and call the superclass constructor with the corresponding $x$ and $y$ values (recall the relations $x = r\cos\theta$ and $y = r\sin\theta$ between Cartesian and polar coordinates). Also, in class `PolarPoint`, add a `__str__` method which prints out $r$, $\theta$, $x$, and $y$ of a point. Verify the implementation by initializing three points and printing these points. Name of program file: `PolarPoint.py`. ◇

**Exercise 9.6.** *Modify a function class by subclassing.*

Consider the `VelocityProfile` class from page 346 for computing the function $v(r; \beta, \mu_0, n, R)$ in formula (4.20) on page 230. Suppose we want to have $v$ explicitly as a function of $r$ and $n$ (this is necessary if we want to illustrate how the velocity profile, the $v(r)$ curve, varies as $n$ varies). We would then like to have a class `VelocityProfile2` that

is initialized with $\beta$, $\mu_0$, and $R$, and that takes $r$ and $n$ as arguments in the `__call__` method. Implement such a class by inheriting from class `VelocityProfile` and by calling the `__init__` and `value` methods in the superclass. It should be possible to try the class out with the following statements:

```
v = VelocityProfile2(beta=0.06, mu0=0.02, R=2)
# evaluate v for various n values at r=0:
for n in 0.1, 0.2, 1:
    print v(0, n)
```

Name of program file: `VelocityProfile2.py`. ◇

**Exercise 9.7.** *Explore the accuracy of difference formulas.*

The purpose of this exercise is to investigate the accuracy of the `Backward1`, `Forward1`, `Forward3`, `Central2`, `Central4`, `Central6` methods for the function[13]

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}} .$$

To solve the exercise, modify the `src/oo/Diff2_examples.py` program which produces tables of errors of difference approximations as discussed at the end of Chapter 9.2.3. Test the approximation methods for $x = 0, 0.9$ and $\mu = 1, 0.01$. Plot the $v(x)$ function for the two $\mu$ values using 1001 points. Name of program file: `boundary_layer_derivative.py`. ◇

**Exercise 9.8.** *Implement a subclass.*

Make a subclass `Sine1` of class `FuncWithDerivatives` from Chapter 9.1.7 for the $\sin x$ function. Implement the function only, and rely on the inherited `df` and `ddf` methods for computing the derivatives. Make another subclass `Sine2` for $\sin x$ where you also implement the `df` and `ddf` methods using analytical expressions for the derivatives. Compare `Sine1` and `Sine2` for computing the first- and second-order derivatives of $\sin x$ at two $x$ points. Name of program file: `Sine12.py`. ◇

**Exercise 9.9.** *Make classes for numerical differentiation.*

Carry out Exercise 7.15. Find the common code in the classes `Derivative`, `Backward`, and `Central`. Move this code to a superclass, and let the three mentioned classes be subclasses of this superclass. Compare the resulting code with the hierarchy shown in Chapter 9.2.1. Name of program file: `numdiff_classes.py`. ◇

**Exercise 9.10.** *Implement a new subclass for differentiation.*

A one-sided, three-point, second-order accurate formula for differentiating a function $f(x)$ has the form

$$f'(x) \approx \frac{f(x-2h) - 4f(x-h) + 3f(x)}{2h} . \tag{9.47}$$

[13] This function is discussed more in detail in Exercise 4.26.

Implement this formula in a subclass `Backward2` of class `Diff` from Chapter 9.2. Compare `Backward2` with `Backward1` for $g(t) = e^{-t}$ for $t = 0$ and $h = 2^{-k}$ for $k = 0, 1, \ldots, 14$ (write out the errors in $g'(t)$). Name of program file: `Backward2.py`. ◇

**Exercise 9.11.** *Understand if a class can be used recursively.*

Suppose you want to compute $f''(x)$ of some mathematical function $f(x)$, and that you apply class `Diff3` from Chapter 9.2.6 twice:

```
ddf = Diff3(Diff3(f, 'central', 2), 'central', 2)
```

Will this work? Hint: Follow the program flow, and find out what the resulting formula will be. Then see if this formula coincides with a formula you know for approximating $f''(x)$. ◇

**Exercise 9.12.** *Represent people by a class hierarchy.*

Classes are often used to model objects in the real world. We may represent the data about a person in a program by a class `Person`, containing the person's name, address, phone number, date of birth, and nationality. A method `__str__` may print the person's data. Implement such a class `Person`.

A worker is a person with a job. In a program, a worker is naturally represented as class `Worker` derived from class `Person`, because a worker *is* a person, i.e., we have an is-a relationship. Class `Worker` extends class `Person` with additional data, say name of company, company address, and job phone number. The print functionality must be modified accordingly. Implement this `Worker` class.

A scientist is a special kind of a worker. Class `Scientist` may therefore be derived from class `Worker`. Add data about the scientific discipline (physics, chemistry, mathematics, computer science, ...). One may also add the type of scientist: theoretical, experimental, or computational. The value of such a type attribute should not be restricted to just one category, since a scientist may be classified as, e.g., both experimental and computational (i.e., you can represent the value as a list or tuple). Implement class `Scientist`.

Researcher, postdoc, and professor are special cases of a scientist. One can either create classes for these job positions, or one may add an attribute (`position`) for this information in class `Scientist`. We adopt the former strategy. When, e.g., a researcher is represented by a class `Researcher`, no extra data or methods are needed. In Python we can create such an "empty" class by writing `pass` (the empty statement) as the class body:

```
class Researcher(Scientist):
    pass
```

Finally, make a demo program where you create and print instances of classes `Person`, `Worker`, `Scientist`, `Researcher`, `Postdoc`, and `Professor`. Print out the attribute contents of each instance (use the `dir` function).

*Remark.* An alternative design is to introduce a class `Teacher` as a special case of `Worker` and let `Professor` be both a `Teacher` and `Scientist`, which is natural. This implies that class `Professor` has two superclasses, `Teacher` and `Scientist`, or equivalently, class `Professor` inherits from to superclasses. This is known as *multiple inheritance* and technically achieved as follows in Python:

```
class Professor(Teacher, Scientist):
    pass
```

It is a continuous debate in computer science whether multiple inheritance is a good idea or not. One obvious problem[14] in the present example is that class `Professor` inherits two names, one via `Teacher` and one via `Scientist` (both these classes inherit from `Person`). Neither of the two widely used languages Java and C# allow multiple inheritance. Nor in this book will we persue the idea of multiple inheritance further. Name of program file: `Person.py`. ◇

**Exercise 9.13.** *Add a new class in a class hierarchy.*

Add the Monte Carlo integration method from Chapter 8.5.1 as a subclass in the `Integrator` hierarchy explained in Chapter 9.3. Import the superclass `Integrator` from the `integrate` module in the file with the new integration class. Test the Monte Carlo integration class in a case with known analytical solution. Name of program file: `MCint_class.py`. ◇

**Exercise 9.14.** *Change the user interface of a class hierarchy.*

All the classes in the `Integrator` hierarchy from Chapter 9.3 take the integration limits $a$ and $b$ plus the number of integration points $n$ as input to the constructor. The `integrate` method takes the function to integrate, $f(x)$, as parameter. Another possibility is to feed $f(x)$ to the constructor and let `integrate` take $a$, $b$, and $n$ as parameters. Make this change to the `integrate.py` file with the `Integrator` hierarchy. Name of program file: `integrate2.py`. ◇

**Exercise 9.15.** *Compute convergence rates of numerical integration methods.*

Most numerical methods have a discretization parameter, call it $n$, such that if $n$ increases (or decreases), the method performs better. Often, the relation between the error in the numerical approximation (compared with the exact analytical result) can be written as

$$E = Cn^r,$$

where $E$ is the error, and $C$ and $r$ are constants.

[14] It is usually not a technical problem, but more a conceptual problem when the world is modeled by objects in a program.

Suppose you have performed an experiment with a numerical method using discretization parameters $n_0, n_1, \ldots, n_N$. You have computed the corresponding errors $E_0, E_1, \ldots, E_N$ in a test problem with an analytical solution. One way to estimate $r$ goes as follows. For two successive experiments we have

$$E_{i-1} = C n_{i-1}^r$$

and

$$E_i = C n_i^r \,.$$

Divide the first equation by the second to eliminate $C$, and then take the logarithm to solve for $r$:

$$r = \frac{\ln(E_{i-1}/E_i)}{\ln(n_{i-1}/n_i)} \,.$$

We can compute $r$ for all pairs of two successive experiments. Usually, the "last $r$", corresponding to $i = N$ in the formula above, is the "best" $r$ value[15]. Knowing this $r$, we can compute $C$ as $E_N n_N^{-r}$.

Having stored the $n_i$ and $E_i$ values in two lists `n` and `E`, the following code snippet computes $r$ and $C$:

```
from scitools.convergencerate import convergence_rate
C, r = convergence_rate(n, E)
```

Construct a test problem for integration where you know the analytical result of the integral. Run different numerical methods (the midpoint method, the Trapezoidal method, Simpson's method, Monte Carlo integration) with the number of evaluation points $n = 2^k + 1$ for $k = 2, \ldots, 11$, compute corresponding errors, and use the code snippet above to compute the $r$ value for the different methods in questions. The higher the absolute error of $r$ is, the faster the method converges to the exact result as $n$ increases, and the better the method is. Which is the best and which is the worst method?

Let the program file import methods from the `integrate` module and the module with the Monte Carlo integration method from Exercise 9.13. Name of program file: `integrators_convergence.py`. ◇

**Exercise 9.16.** *Add common functionality in a class hierarchy.*

Suppose you want to use classes in the `Integrator` hieararchy from Chapter 9.3 to calculate integrals of the form

$$F(x) = \int_a^x f(t)dt \,.$$

[15] This guideline is rough. If the method convergences, and round-off errors do not influence the values of $E_i$, the guideline is good. However, for very large/small $n$ round-off errors can cause the method to diverge, and then the "last $r$" is not a relevant value to pick.

Such functions $F(x)$ can be efficiently computed by the method from Exercise 7.22. Implement this computation of $F(x)$ in an additional method in the superclass `Integrator`. Test that the implementation is correct for $f(x) = 2x - 3$ for all the implemented integration methods (the Midpoint, Trapezoidal and Gauss-Legendre methods, as well as Simpson's rule, integrate a linear function exactly). Name of program file: `integrate_efficient.py`. ◇

**Exercise 9.17.** *Make a class hierarchy for root finding.*

Given a general nonlinear equation $f(x) = 0$, we want to implement classes for solving such an equation, and organize the classes in a class hierarchy. Make classes for three methods: Newton's method (Chapter 5.1.9), the Bisection method (Chapter 3.6.2), and the Secant method (Exercise 5.14). Move common code (starting values, the $f(x)$ functions, parameters in termination criteria, etc.) to a common superclass. Do Exercise 5.15 using the new class hierarchy. Name of program file: `Rootfinders.py`. ◇

**Exercise 9.18.** *Use the ODESolver hierarchy to solve a simple ODE.*

Solve the ODE problem $u' = u/2$ with $u(0) = 1$, using a class in the `ODESolver` hierarchy. Choose $\Delta t = 0.5$ and perform $N = 12$ steps. Write out the approximate $u_N$ together with the exact value $e^{N\Delta t/2}$. Name of program file: `ODESolver_demo.py`. ◇

**Exercise 9.19.** *Use the 4th-order Runge-Kutta on (B.34).*

Investigate if the 4th-order Runge-Kutta method is better than the Forward Euler scheme for solving the challenging ODE problem (B.34) from Exercise B.3 on page 621. Name of program file: `yx_ODE2.py`. ◇

**Exercise 9.20.** *Solve an ODE until constant solution.*

Newton's law of cooling,

$$\frac{dT}{dt} = -h(T - T_s) \tag{9.48}$$

can be used to see how the temperature $T$ of an object changes because of heat exchange with the surroundings, which have a temperature $T_s$. The parameter $h$, with unit $s^{-1}$ is an experimental constant (heat transfer coefficient) telling how efficient the heat exchange with the surroundings is. For example, (9.48) may model the cooling of a hot pizza taken out of the oven. The problem with applying (9.48), nevertheless, is that $h$ must be measured. Suppose we have measured $T$ at $t = 0$ and $t_1$. We can use a rough Forward Euler approximation of (9.48) with one time step of length $t_1$,

$$\frac{T(t_1) - T(0)}{t_1} = -h(T(0) - T_s),$$

to make the estimate

$$h = \frac{T(t_1) - T(0)}{t_1(T_s - T(0))} . \qquad (9.49)$$

Suppose now you take a hot pizza out of the oven. The temperature of the pizza is 200 C at $t = 0$ and 180 C after 20 seconds, in a room with temperature 20 C. Find an estimate of $h$ from the formula above.

Solve (9.48) to find the evolution of the temperature of the pizza. Use class `ForwardEuler` or `RungeKutta4`, and supply a `terminate` function to the `solve` method so that the simulation stops when $T$ is sufficiently close to the final room temperature $T_s$. Plot the solution. Name of program file: `pizza_cooling1.py`. ◇

**Exercise 9.21.** *Use classes in Exer. 9.20.*

Solve Exercise 9.20 with a class `Problem` containing the parameters $h$, $T_s$, $T(0)$, $t_1$, and $T(t_1)$ as attributes. The class should have a method `estimate_h` for returning an estimate of $h$, given the other parameters. Also a method `__call__` for computing the right-hand side must be included. The `terminate` function can be a method in the class as well. By using class `Problem`, we avoid having the physical parameters as global variables in the program. Name of program file: `pizza_cooling2.py`. ◇

**Exercise 9.22.** *Scale away parameters in Exer. 9.20.*

Use the scaling approach from Chapter 9.4.7 to "scale away" the parameters in the ODE in Exercise 9.20. That is, introduce a new unknown $u = (T - T_s)/(T(0) - T_s)$ and a new time scale $\tau = th$. Find the ODE and the initial condition that governs the $u(\tau)$ function. Make a program that computes $u(\tau)$ until $|u| < 0.001$. Store the discrete $u$ and $\tau$ values in a file `u_tau.dat` if that file is not already present (you can use `os.path.isfile(f)` to test if a file with name `f` exists). Create a function `T(u, tau, h, T0, Ts)` that loads the $u$ and $\tau$ data from the `u_tau.dat` file and returns two arrays with $T$ and $t$ values, corresponding to the computed arrays for $u$ and $\tau$. Plot $T$ versus $t$. Give the parameters $h$, $T_s$, and $T(0)$ on the command line. Note that this program is supposed to solve the ODE once and then recover any $T(t)$ solution by a simple scaling of the single $u(\tau)$ solution. Name of program file: `pizza_cooling3.py`. ◇

**Exercise 9.23.** *Compare ODE methods.*

Equation (7.6) is a relevant model for radioactive decay. The function $u(t)$ is the fraction of particles that remains in the radioactive substance at time $t$. The parameter $a$ is the inverse of the so-called mean lifetime of the substance. The initial condition is $u(0) = 1$.

Introdce a class `Decay` to hold information about the physical problem: the parameter $a$ and a `__call__` method for computing the right-hand side $-au$ of the ODE (see Chapters 7.4.4 or 9.4.8 for examples). Initialize an instance of class `Decay` with $a = \ln(2)/5600$ 1/years (this value of $a$ corresponds to the Carbon-14 radioactive isotope whose

decay is used extensively in dating organic material that is tens of thousands of years old).

Solve (7.6) by both the Forward Euler and the 4-th order Runge-Kutta method, using the `ForwardEuler` and the `RungeKutta4` classes in the `ODESolver` hierarchy. Use a time step of 500 years, and simulate decay for 20,000 years (let the time unit be 1 year). Plot the two solutions. Write out the final $N$ value from the simulations and compare it with the exact solution $N(t) = N_0 e^{-N\Delta t/\tau}$. Name of program file: `radioactive_decay.py`. ◇

**Exercise 9.24.** *Solve two coupled ODEs for radioactive decay.*

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a mean lifetime $\tau_A$, while substance B decay to form type A nuclei with a mean lifetime $\tau_B$. Letting $u_A$ and $u_B$ be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$u'_A = u_B/\tau_B - u_A/\tau_A, \tag{9.50}$$

$$u'_B = u_A/\tau_A - u_B/\tau_B, \tag{9.51}$$

with $u_A(0) = u_B(0) = 1$. As in Exercise 9.23, introduce a problem class, which holds the parameters $\tau_A$ and $\tau_B$ and offers a `__call__` method to compute the right-hand side vector of the ODE system, i.e., $(u_B/\tau_B - u_A/\tau_A, u_A/\tau_A - u_B/\tau_B)$. Solve for $u_A$ and $u_B$ using a subclass in the `ODESolver` hierarchy and the parameter choice $\tau_A = 8$ minutes, $\tau_B = 40$ minutes, and $\Delta t = 10$ seconds. Plot $u_A$ and $u_B$ against time measured in minutes. From the ODE system it follows that the ratio $u_A/u_B \rightarrow \tau_A/\tau_B$ as $t \rightarrow \infty$ (asssuming $u'_A = u'_B = 0$ in the limit $t \rightarrow \infty$). Check that the solutions fulfills this requirement (this is a partial verification of the program). Name of program file: `radioactive_decay2.py`. ◇

**Exercise 9.25.** *Compare methods for solving the ODE (B.36).*

Consider emptying a tank of water as described in Exercise B.7 on page 554. The governing ODE problem is (B.36) with $h(0) = h_0$. Make a class `Tank` for storing the physical parameters of the problem: $r$, $R$, $g$, and $h_0$. This class should also have a `__call__` method for defining the right-hand side of (B.36). Solve this ODE problem using the `ForwardEuler`, `BackwardEuler`, and `RungeKutta4` classes in the `ODESolver` hierarchy. Apply data in the `Tank` instance to initialize the solver classes. Read $\Delta t$ from the command line and try out values between 5 and 50 s. Compare the numerical solutions in a plot. Comment upon the quality of the various methods to compute a correct limiting value of $h$ (0) as $\Delta t$ is varied. Name of program file: `tank_ODE_3methods.py`. ◇

**Exercise 9.26.** *Code a 2nd-order Runge-Kutta method; function.*

Implement the 2nd-order Runge-Kutta method specified in formula (9.25) for solving ordinary differential equations. Use a plain function `RungeKutta2` of the type shown in Chapter 7.4.1 for the Forward Euler method. Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Name of program file: `RungeKutta2_func.py`. ◇

**Exercise 9.27.** *Code a 2nd-order Runge-Kutta method; class.*

Make a new subclass `RungeKutta2` in the `ODESolver` hierarchy from Chapter 9.4 for solving ordinary differential equations with the 2nd-order Runge-Kutta method specified in formula (9.25). Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Store the `RungeKutta2` class and the test problem in a file where the base class `ODESolver` is imported from the `ODESolver` module. Name of program file: `RungeKutta2.py`. ◇

**Exercise 9.28.** *Implement a midpoint method for ODEs.*

This exercise is similar to Exercise 9.27, but the purpose now is to implement the midpoint method specified in formula (9.24).

Compare in a plot the midpoint method with the Forward Euler and 4th-order Runge-Kutta methods and the exact solution for the problem $u' = u$, $u(0) = 1$, with $\Delta t = 0.5$ and 10 steps. Name of program file: `Midpoint.py`. ◇

**Exercise 9.29.** *Implement a modified Euler method for ODEs.*

Do Exercise 7.29 and incorporate the class in the `ODESolver` hierarchy. Compare the method with other methods in the same test problem as in Exercise 9.28. Name of program file: `ModifiedEuler.py`. ◇

**Exercise 9.30.** *Improve the implementation in Exer. 7.25.*

We consider the physical problem of an object falling or rising in a fluid as described in Exercise 7.25. The purpose now is to solve the governing ODE (7.16) using classes in the`ODESolver` hierarchy. We also want to set the physical and numerical parameters of the problem on the command line.

First make a class for defining the right-hand side of (7.16). The physical parameters needed in the definition of the right-hand side should be attributes in the class.

Continue with making a function `solve(method, f, v0, T, dt)` for solving (7.16). The argument `method` is the name of a subclass in the `ODESolver` hieararchy, `f` is the object defining the right-hand side of the ODE, `v0` is the initial condition, `T` is the final time for the simulation, and `dt` is the time step. The `solve` should return two arrays, one with the velocity values and one with the corresponding time values.

A separate function `read_input` can read all the input data from the command line, preferably using the `getopt` module, or better, the

`ReadInput` class hierarchy from Chapter 9.6.2. The `read_input` function first sets some default values, then reads input, and finally returns the set of variables that must be sent further to the `solve` function (see `src/box_spring/box_spring.py` for a similar example using `getopt`).

Implement the simple verification test as a function `verify`. The two other real test cases can be implemented in two separate functions, `parachute_jumper` and `rising_ball`. Let these three functions return the computed velocities and corresponding time points. Collect the right-hand side class and all the functions in a module file. Run one of the three cases from the test block, using a command-line argument to determine which case. Name of program file: `body_in_fluid2.py`. ◇

**Exercise 9.31.** *Visualize the different forces in Exer. 9.30.*

The purpose of this exercise is to plot the forces $F_g$, $F_b$, and $F_d$ in the model from Exercise 9.30 as functions of $t$. Seeing the relative importance of the forces as time develops gives an increased understanding of how the different forces contribute to change the velocity.

Import the functions from the module developed in Exercise 9.30 in a new program, call the function for computing one of the two real cases from Exercise 9.30, and feed the returned $v$ to a new function for computing the forces $F_g$, $F_b$, and $F_d$. Plot these three forces against time. Name of program file: `body_in_fluid_forces.py`. ◇

**Exercise 9.32.** *Find the body's position in Exer. 9.30.*

In Exercise 9.30 we compute $v(t)$. The position of the body, $y(t)$, is related to the velocity $v$ by $y'(t) = v(t)$. Extend the program from Exercise 9.30 to solve the system

$$
\begin{aligned}
\frac{dy}{dt} &= v,\\
\frac{dv}{dt} &= -g\left(1-\frac{\varrho}{\varrho_b}\right) - -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,.
\end{aligned}
$$

Name of program file: `body_in_fluid2.py`. ◇

**Exercise 9.33.** *Compare methods for solving (B.37)–(B.38).*

Consider the system of ODEs in Exercise B.8 for simulating an electric circuit. The purpose now is to compare the Forward Euler scheme with the 4-th order Runge-Kutta method. Make a class `Circuit` for storing the physical parameters of the problem ($L$, $R$, $C$, $E(t)$) as well as the initial conditions ($I(0)$, $Q(0)$). Class `Circuit` should also define the right-hand side of the ODE through a `__call__` method. Create two solver instances, one from the `ForwardEuler` class and one from the `RungeKutta4` class. Solve the ODE system using both methods. Plot the two $I(t)$ solutions for comparison. As you will see, the Forward Euler scheme overestimates the amplitudes significantly, compared with the more accurate 4th-order Runge-Kutta method. Name of program file: `electric_circuit2.py`. ◇

**Exercise 9.34.** *Add the effect of air resistance on a ball.*

The differential equations governing the horizontal and vertical motion of a ball subject to gravity and air resistance read[16]

$$\frac{d^2x}{dt^2} = -\frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dx}{dt}, \tag{9.52}$$

$$\frac{d^2y}{dt^2} = -g - \frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dy}{dt}, \tag{9.53}$$

where $(x, y)$ is the position of the ball ($x$ is a horizontal measure and $y$ is a vertical measure), $g$ is the acceleration of gravity, $C_D = 0.2$ is a drag coefficient, $\bar{\varrho}$ is the ratio of the density of air and the ball, and $a$ is the radius of the ball. The latter two quantities can be taken as 0.017 and 11 cm for a football.

Let the initial condition be $x = y = 0$ (start position in origo) and

$$dx/dt = v_0 \cos\theta, \quad dy/dt = v_0 \sin\theta,$$

where $v_0$ is the magnitude of the initial velocity and $\theta$ is the angle the velocity makes with the horizontal. For a hard football kick we can set $v_0 = 120$ km/h and take $\theta$ as 30 degrees.

Express the two second-order equations above as a system of four first-order equations with four initial conditions. Implement the right-hand side in a problem class where the physical parameters $C_D$, $\bar{\varrho}$, $a$, $v_0$, and $\theta$ are stored along with the initial conditions.

Solve the ODE system for $C_D = 0$ (no air resistance) and $C_D = 0.2$, and plot $y$ as a function of $x$ in both cases to illustrate the effect of air resistance. Use the 4-th order Runge-Kutta method. (Make sure you express all units in kg, m, s, and radians.) Name of program file: `kick2D.py`. ◇

**Exercise 9.35.** *Make a class for drawing an arrow.*

Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing an arrow. An arrow consists of three lines, so the arrow class will naturally contain three `Line` instances. Let each line in the arrow head make an angle of 30 degrees with the main line, and let the length of the arrow head be 1/8 of the length of the main line. It is easiest to always draw a vertical arrow in the `Arrow` class. The constructor can then take a bottom point and the length of the arrow. With the `rotate` method the user can later rotate the arrow.

Make some arrows of different lengths, and call `rotate` to rotate them differently. Name of program file: `Arrow.py`. ◇

[16] The equations arise by combining the models in Exercises 1.10 and 1.14.

**Exercise 9.36.** *Make a class for drawing a person.*

A very simple sketch of a human being can be made of a circle for the head, two lines for the arms, one vertical line or a rectangle for the torso, and two lines for the legs. Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing such a simple sketch of a person. Build the figure from `Line` and `Circle` instances. Supply the following arguments to the constructor: the center point of the head and the radius $R$ of the head. Let the arms and the torso be of length $4R$, and the legs of length $6R$. The angle between the legs can be fixed (say 30 degrees), while the angle of the arms relative to the torso can be an argument to the constructor with a suitable default value. Name of program file: `draw_person.py`. ◇

**Exercise 9.37.** *Animate a person with waving hands.*

Make a subclass of the class from Exercise 9.36 where the constructor can take an argument describing the angle between the arms and the torso. Use this new class to animate a person who waves her/his hands. Name of program file: `draw_waving_person.py`. ◇

**Exercise 9.38.** *Make a class for drawing a car.*

A two-dimensional car can be drawn by putting together a rectangle, circles, arcs, and lines. Make a class in the `Shape` hierarchy from Chapter 9.5 for drawing a car, following the same principle as in Exercise 9.36. The constructor takes a length $L$ of the car and the coordinates of a point $p$. The various shapes that build up the car must have dimensions relative to $L$ and placements relative to $p$. Draw a small car and a large car in the same figure. Name of program file: `draw_car.py`. ◇

**Exercise 9.39.** *Make a car roll.*

Use the class for drawing a car in Exercise 9.38 and the ideas from Chapter 9.5.5 to make an animation of a rolling car. Implement the rolling functionality in a subclass of the car class from Exercise 9.38. Name of program file: `rolling_car.py`. ◇

**Exercise 9.40.** *Make a class for differentiating noisy data.*

Suppose you have some time series signal $y(t_k)$ for $k = 0, \ldots, n-1$, where $t_k = k\Delta t$ are time points. Differentiating such a signal can give very inaccurate results if the signal contains noise. Exercises 8.44–8.47 explore this topic, and Exericse 8.47 suggests to filter the signal. The purpose of the present exercise is to make a tool for differentating noisy signals.

Make a class `DiffNoisySignal` where the constructor takes three arguments: the signal $y(t_k)$ (as an array), the order of the desired derivative (as an `int`, either 1 or 2), and the name of the signal (as a string). A method `filter(self, n)` runs the filter from Exericse 8.47 `n` times on the signal. The method `diff(self)` performs the differentiation and

stores the differentiated signal as an attribute in the class. There should also be some plotting methods: `plot(self)` for plotting the current (original or filtered) signal, `plot_diff(self)` for plotting the differentiated signal, `animate_filter` for animating the effect of filtering (run `filter` once per frame in the movie), and `animate_diff` for animating the evolution of the derivative when `filter` and `diff` are called once each per frame.

Implement the class and test it on the noisy signal

$$y(t_k) = \cos(2\pi t_k) + 0.1r_k, \quad t_k = k\Delta t,\ k = 0, \ldots, n-1,$$

with $\Delta t = 1/60$. The quantities $r_k$ are random numbers in $[0, 1)$. Make animations with the `animate_filter` and `animate_diff` methods. Name of program file: `DiffNoisySignal.py`. ◇

**Exercise 9.41.** *Find local and global extrema of a function.*

Extreme points of a function $f(x)$ are normally found by solving $f'(x) = 0$. A much simpler method is to evaluate $f(x)$ for a set of discrete points in the interval $[a, b]$ and look for local minima and maxima among these points. We work with $n$ equally spaced points $a = x_0 < x_1 < \cdots < x_{n-1} = b$, $x_i = a + ih$, $h = (b-a)/(n-1)$.

1. First we find all local extreme points in the interior of the domain. Local minima are recognized by

   $$f(x_{i-1}) > f(x_i) < f(x_{i+1}), \quad i = 1, \ldots, n-2\,.$$

   Similarly, at a local maximum point $x_i$ we have

   $$f(x_{i-1}) < f(x_i) > f(x_{i+1}), \quad i = 1, \ldots, n-2\,.$$

   We let $P_{\min}$ be the set of $x$ values for local minima and $F_{\min}$ the set of the corresponding $f(x)$ values at these minimum points. Two sets $P_{\max}$ and $F_{\max}$ are defined correspondingly, containing the maximum points and their values.
2. The boundary points $x = a$ and $x = b$ are for algorithmic simplicity also defined as local extreme points: $x = a$ is a local minimum if $f(a) < f(x_1)$, and a local maximum otherwise. Similarly, $x = b$ is a local minimum if $f(b) < f(x_{n-2})$, and a local maximum otherwise. The end points $a$ and $b$ and the corresponding function values must be added to the sets $P_{\min}, P_{\max}, F_{\min}, F_{\max}$.
3. The global maximum point is defined as the $x$ value corresponding to the maximum value in $F_{\max}$. The global minimum point is the $x$ value corresponding to the minimum value in $F_{\min}$.

Make a class `MinMax` with the following functionality:

- The constructor takes $f(x)$, $a$, $b$, and $n$ as arguments, and calls a method `_find_extrema` to compute the local and global extreme points.
- The method `_find_extrema` implements the algorithm above for finding local and global extreme points, and stores the sets $P_{\rm min}, P_{\rm max}, F_{\rm min}, F_{\rm max}$ as list attributes in the (`self`) instance.
- The method `get_global_minimum` returns the global minimum point ($x$).
- The method `get_global_maximum` returns the global maximum point ($x$).
- The method `get_all_minima` returns a list or array of all minimum points.
- The method `get_all_maxima` returns a list or array of all maximum points.
- The method `__str__` returns a string where all the min/max points are listed, plus the global extreme points.

Here is a sample code using class `MinMax`:

```
def f(x):
    return x**2*exp(-0.2*x)*sin(2*pi*x)

m = MinMax(f, 0, 4)
print m
```

The output becomes

```
All minima: 0.8056, 1.7736, 2.7632, 3.7584, 0
All maxima: 0.3616, 1.284, 2.2672, 3.2608, 4
Global minimum: 3.7584
Global maximum: 3.2608
```

Make sure that the program also works for functions without local extrema, e.g., linear functions $f(x) = px + q$. Name of program file: `minmaxf.py`. ◇

**Exercise 9.42.** *Improve the accuracy in Exer. 9.41.*

The algorithm in Exercise 9.41 finds a local extreme point if the function value at $x_i$ is larger or smaller than the function values at the neighboring points. If we have found a minimum at $x_i$, all we know is that $f(x)$ has a minimum in the interval $(x_{i-1}, x_{i+1})$. With $h$ as the distance between the points, the error in the coordinate of an extreme point can be as high as $2h$. Of course, increasing the number of points decreases this error. Nevertheless, we may think of a computationally more efficient method, namely a bisection method for finding $f'(x) = 0$ in $(x_{i-1}, x_{i+1})$. In class `MinMax`, add a method `_refine_extrema`, which goes through all the interior local minima and maxima, makes a callable object for $f'(x)$ using the `Central2` class in the `Diff` hierarchy, and calls a bisection method to find where $f'(x) = 0$. The tolerance in the termination criterion for the bisection method can be given as an argument to the constructor (`None` signifies that no bisection algorithm is applied). Name of program file: `minmaxf2.py`. ◇

**Exercise 9.43.** *Make a calculus calculator class.*

Given a function $f(x)$ defined on a domain $[a, b]$, the purpose of many mathematical exercises is to sketch the function curve $y = f(x)$, compute the derivative $f'(x)$, find local and global extreme points, and compute the integral $\int_a^b f(x)dx$. Make a class `CalculusCalculator` which can perform all these actions for any function $f(x)$ using numerical differentiation and integration, and the method explained in Exercise 9.41 or 9.42 for finding extrema.

Here is an interactive session with the class where we analyze $f(x) = x^2 e^{-0.2x} \sin(2\pi x)$ on $[0, 6]$ with a grid (set of $x$ coordinates) of 700 points:

```
>>> from CalculusCalculator import *
>>> def f(x):
...     return x**2*exp(-0.2*x)*sin(2*pi*x)
...
>>> c = CalculusCalculator(f, 0, 6, resolution=700)
>>> c.plot()                 # plot f
>>> c.plot_derivative()      # plot f'
>>> c.extreme_points()

All minima: 0.8052, 1.7736, 2.7636, 3.7584, 4.7556, 5.754, 0
All maxima: 0.3624, 1.284, 2.2668, 3.2604, 4.2564, 5.2548, 6
Global minimum: 5.754
Global maximum: 5.2548

>>> c.integral
-1.7353776102348935
>>> c.df(2.51)    # c.df(x) is the derivative of f
-24.056988888465636
>>> c.set_differentiation_method(Central4)
>>> c.df(2.51)
-24.056988832723189
>>> c.set_integration_method(Simpson)  # more accurate integration
>>> c.integral
-1.7353857856973565
```

Design the class such that the above session can be carried out.

Hint: Use classes from the `Diff` and `Integrator` hierarchies (Chapters 9.2 and 9.3) for numerical differentiation and integration (with, e.g., `Central2` and `Trapezoidal` as default methods for differentiation and integration, respectively). The method `set_differentiation_method` takes a subclass name in the `Diff` hierarchy as argument, and makes an attribute `df` that holds a subclass instance for computing derivatives. With `set_integration_method` we can similarly set the integration method as a subclass name in the `Integrator` hierarchy, and then compute the integral $\int_a^b f(x)dx$ and store the value in the attribute `integral`. The `extreme_points` method performs a `print` on a `MinMax` instance, which is stored as an attribute in the calculator class. Name of program file: `CalculusCalculator.py`.

◇

**Exercise 9.44.** *Extend Exer. 9.43.*

Extend class `CalculusCalculator` from Exercise 9.43 to offer computations of inverse functions. A numerical way of computing inverse functions is explained in Chapter 5.1.10. Exercise 7.20 suggests an improved implementation using classes. Use the `InverseFunction` implementation from Exercise 7.20 in class `CalculusCalculator`. Name of program file: `CalculusCalculator2.py`. ◇

**Exercise 9.45.** *Formulate a 2nd-order ODE as a system.*

In this and subsequent exercises we shall deal with the following second-order ordinary differential equation with two initial conditions:

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad t > 0, \quad u(0) = U_0,\ \dot{u}(0) = V_0\,. \tag{9.54}$$

Write (9.54) as a system of two first-order differential equations. Also set up the initial condition for this system.

*Physical Applications.* Equation (9.54) has a wide range of applications throughout science and engineering. A primary application is damped spring systems in, e.g., cars and bicycles: $u$ is the vertical displacement of the spring system attached to a wheel; $\dot{u}$ is then the corresponding velocity; $F(t)$ resembles a bumpy road; $s(u)$ represents the force from the spring; and $f(\dot{u})$ models the damping force (friction) in the spring system. For this particular application $f$ and $s$ will normally be linear functions of their arguments: $f(\dot{u}) = \beta\dot{u}$ and $s(u) = ku$, where $k$ is a spring constant and $\beta$ some parameter describing viscous damping.

Equation (9.54) can also be used to describe the motions of a moored ship or oil platform in waves: the moorings act as a nonlinear spring $s(u)$; $F(t)$ represents environmental excitation from waves, wind, and current; $f(\dot{u})$ models damping of the motion; and $u$ is the one-dimensional displacement of the ship or platform.

Oscillations of a pendulum can be described by (9.54): $u$ is the angle the pendulum makes with the vertical; $s(u) = (mg/L)\sin(u)$, where $L$ is the length of the pendulum, $m$ is the mass, and $g$ is the acceleration of gravity; $f(\dot{u}) = \beta|\dot{u}|\dot{u}$ models air resistance (with $\beta$ being some suitable constant, see Exercises 1.10 and 9.50); and $F(t)$ might be some motion of the top point of the pendulum.

Another application is electric circuits with $u(t)$ as the charge, $m = L$ as the inductance, $f(\dot{u}) = R\dot{u}$ as the voltage drop accross a resistor $R$, $s(u) = u/C$ as the voltage drop accross a capacitor $C$, and $F(t)$ as an electromotive force (supplied by a battery or generator).

Furthermore, Equation (9.54) can act as a simplified model of many other oscillating systems: aircraft wings, lasers, loudspeakers, microphones, tuning forks, guitar strings, ultrasound imaging, voice, tides, the El Niño phenomenon, climate changes – to mention some.

We remark that (9.54) is a possibly nonlinear generalization of Equation (C.8) explained in Appendix C.1.3. The case in Appendix C cor-

responds to the special choice of $f(\dot{u})$ proportional to the velocity $\dot{u}$, $s(u)$ proportional to the displacement $u$, and $F(t)$ as the acceleration $\ddot{w}$ of the plate and the action of the gravity force. ◇

**Exercise 9.46.** *Solve the system in Exer. 9.45 in a special case.*

Make a function

```
def rhs(u, t):
    ...
```

for returning the right-hand side of the first-order differential equation system from Exercise 9.45. As usual, the `u` argument is an array or list with the two solution components `u[0]` and `u[1]` at some time `t`. Inside `rhs`, assume that you have access to three global Python functions `friction(dudt)`, `spring(u)`, and `external(t)` for evaluating $f(\dot{u})$, $s(u)$, and $F(t)$, respectively.

Test the `rhs` function in combination with the functions $f(\dot{u}) = 0$, $F(t) = 0$, $s(u) = u$, and the choice $m = 1$. The differential equation then reads $\ddot{u} + u = 0$. With initial conditions $u(0) = 1$ and $\dot{u}(0) = 0$, one can show that the solution is given by $u(t) = \cos(t)$. Apply two numerical methods: the 4th-order RungeKutta method and the Forward Euler method from the `ODESolver` module developed in Chapter 9.4. Use a time step $\Delta t = \pi/20$.

Plot $u(t)$ and $\dot{u}(t)$ versus $t$ together with the exact solutions. Also make a plot of $\dot{u}$ versus $u$ (`plot(u[:,0], u[:,1])` if `u` is the array returned from the solver's `solve` method). In the latter case, the exact plot should be a circle[17], but the ForwardEuler method results in a spiral. Investigate how the spiral develops as $\Delta t$ is reduced.

The kinetic energy $K$ of the motion is given by $\frac{1}{2}m\dot{u}^2$, and the potential energi $P$ (stored in the spring) is given by the work done by the spring force: $P = m\int_0^u s(v)dv = \frac{1}{2}mu^2$. Make a plot with $K$ and $P$ as functions of time for both the 4th-order Runge-Kutta method and the Forward Euler method. In the present test case, the sum of the kinetic and potential energy should be constant. Compute this constant analytically and plot it together with the sum $K + P$ as computed by the 4th-order Runge-Kutta method and the Forward Euler method.

Name of program file: `oscillator_v1.py`. ◇

**Exercise 9.47.** *Enhance the code from Exer. 9.46.*

The `rhs` function written in Exercise 9.46 requires that threre is one particular set of Python functions `friction(dudt)`, `spring(u)`, and `external(t)` representing $f(\dot{u})$, $s(u)$, and $F(t)$, respectively. One must also assume that a global variable `m` holds the value of $m$. Frequently, we want to work with different choices of $f(\dot{u})$, $s(u)$, and $F(t)$, which with the `rhs` function proposed in Exercise 9.46 leads to `if` tests for

[17] The points on the curve are $(\cos t, \sin t)$, which all lie on a circle as $t$ is varied.

the choices inside the `friction`, `spring`, and `external` functions. For example,

```
def spring(u):
    if spring_type == 'linear':
        return k*u
    elif spring_type == 'cubic':
        return k*(u - 1./6*u**3)
```

It would be better to introduce two different `spring` functions instead, or represent these functions by classes as explained in Chapter 7.1.2.

Instead of the `rhs` function in Exercise 9.46, develop a class `RHS` where the constructor takes the $f(\dot{u})$, $s(u)$, and $F(t)$ functions as arguments `friction`, `spring`, and `external`. The $m$ parameter must also be an argument. Use a `__call__` method to evaluate the right-hand side of the differential equation system arising from (9.54).

Write a function

```
def solve(T,
          dt,
          initial_u,
          initial_dudt,
          method=RungeKutta4,
          m=1.0,
          friction=lambda dudt: 0,
          spring=lambda u: u,
          external=lambda t: 0):
    ...
    return u, t
```

for solving (9.54) from time zero to some stopping time `T` with time step `dt`. The other arguments hold the initial conditions for $u$ and $\dot{u}$, the class for the numerical solution method, as well as the $f(\dot{u})$, $s(u)$, and $F(t)$ functions. (Note the use of lambda functions, see Chapter 2.2.11, to quickly define some default choices for $f(\dot{u})$, $s(u)$, and $F(t)$). The `solve` function must create an `RHS` instance and feed this to an instance of the class referred to by `method`.

Also write a function

```
def makeplot(T,
             dt,
             initial_u,
             initial_dudt,
             method=RungeKutta4,
             m=1.0,
             friction=lambda dudt: 0,
             spring=lambda u: u,
             external=lambda t: 0,
             u_exact=None):
```

which calls `solve` and makes plots of $u$ versus $t$, $\dot{u}$ versus $t$, and $\dot{u}$ versus $u$. If `u_exact` is not `None`, this argument holds the exact $u(t)$ solution, which should then be included in the plot of the numerically computed solution.

Make a function `get_input`, which reads input data from the command line and calls `makeplot` with these data. Use option-value pairs on the command line to specify `T`, `dt`, `initial_u`, `initial_dudt`, `m`, `method`, `friction`, `spring`, `external`, and `u_exact`. Use `eval` on the first five values so that mathematical expressions like `pi/10` can be specified. Also use `eval` on `method` to transform the string with a class name into a Python class object. For the latter four arguments, assume the command-line value is a string that can be turned into a function via the `StringFunction` tool from Chapter 3.1.4. Let string formulas for `friction`, `spring`, and `external` have `dudt`, `u`, and `t` as independent variables, respectively. For example,

```
elif option == '--friction':
    friction = StringFunction(value, independent_variable='dudt')
```

The `friction`, `spring`, and `external` functions will be called with a scalar (real number) argument, while it is natural to call `u_exact` with an array of time values. In the latter case, the `StringFunction` object must be vectorized (see Chapter 4.4.3):

```
elif option == '--u_exact':
    u_exact = StringFunction(value, independent_variable='t')
    u_exact.vectorize(globals())
```

Collect the functions in a module, and let the test block in this module call the `get_input` function. Test the module by running the tasks from Exercise 9.46:

Terminal

```
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
                 --dt "pi/20" --T "5*pi"
oscillator_v2.py --method RungeKutta4  --u_exact "cos(t)" \
                 --dt "pi/20" --T "5*pi"
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
                 --dt "pi/40" --T "5*pi"
oscillator_v2.py --method ForwardEuler --u_exact "cos(t)" \
                 --dt "pi/80" --T "5*pi"
```

A demo with friction and external forcing can also be made, for example,

Terminal

```
oscillator_v2.py --method RungeKutta4 --friction "0.1*dudt" \
        --external "sin(0.5*t)" --dt "pi/80" --T "40*pi" --m 10
```

Name of program file: `oscillator_v2.py`. ◇

**Exercise 9.48.** *Make a tool for analyzing oscillatory solutions.*

The solution $u(t)$ of the equation (9.54) often exhibit an oscillatory behaviour (for the test problem in Exercise 9.46 we have that $u(t) = \cos t$). It is then of interest to find the wavelength of the oscillations. The purpose of this exercise is to find and visualize the distance between peaks in a numerical representation of a continuous function.

Given an array $(y_0, \ldots, y_{n-1})$ representing a function $y(t)$ sampled at various points $t_0, \ldots, t_{n-1}$. A local maximum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} < y_k > y_{k+1}$. Similarly, a local minimum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} > y_k < y_{k+1}$. By iterating over the $y_1, \ldots, y_{n-2}$ values and making the two tests, one can collect local maxima and minima as $(t_k, y_k)$ pairs. Make a function `minmax(t, y)` which returns two lists, `minima` and `maxima`, where each list holds pairs (2-tuples) of $t$ and $y$ values of local minima or maxima. Ensure that the $t$ value increases from one pair to the next. The arguments `t` and `y` in `minmax` hold the coordinates $t_0, \ldots, t_{n-1}$ and $y_0, \ldots, y_{n-1}$, respectively.

Make another function `wavelength(peaks)` which takes a list `peaks` of 2-tuples with $t$ and $y$ values for local minima or maxima as argument and returns an array of distances between consequtive $t$ values, i.e., the distances between the peaks. These distances reflect the local wavelength of the computed $y$ function. More precisely, the first element in the returned array is `peaks[1][0]-peaks[0][0]`, the next element is `peaks[2][0]-peaks[1][0]`, and so forth.

Test the `minmax` and `wavelength` functions on $y$ values generated by $y = e^{t/4}\cos(2t)$ and $y = e^{-t/4}\cos(t^2/5)$ for $t \in [0, 4\pi]$. Plot the $y(t)$ curve in each case, and mark the local minima and maxima computed by `minmax` with circles and boxes, respectively. Make a separate plot with the array returned from the `wavelength` function (just plot the array against its indices - the point is to see if the wavelength varies or not). Plot only the wavelengths corresponding to maxima.

Make a module with the `minmax` and `wavelength` function, and let the test block perform the tests specified above. Name of program file: `wavelength.py`. ◇

**Exercise 9.49.** *Replace functions by class in Exer. 9.46.*

The three functions `solve`, `makeplot`, and `get_input` from Exercise 9.46 contain a lot of arguments. Instead of shuffling long argument lists into functions, we can create classes that hold the arguments as attributes.

Introduce three classes: `Problem`, `Solver`, and `Visualize`. In class `Problem`, we store the specific data about the problem to be solved, in this case the parameters `initial_u`, `initial_dudt`, `m`, `friction`, `spring`, `external`, and `u_exact`, using the namings in Exercise 9.46. Methods can read the user's values from the command line and initialize attributes, and form the right-hand side of the differential equation system to be solved.

In class `Solver`, we store the data related to solving a system of ordinary differential equations: `T`, `dt`, and `method`, plus the solution. Methods can read input from the command line and initialize attributes, and solve the ODE system using information from a class `Problem` instance.

The final class, `Visualize`, has attributes holding the solution of the problem and can make various plots. We may control the type of plots by command-line arguments.

Class `Problem` may look like

```
class Problem:
    def initialize(self):
        """Read option-value pairs from sys.argv."""
        self.m = eval(read_cml('--m', 1.0))
        ...
        s = read_cml('--spring', '0')
        self.spring = StringFunction(s, independent_variable='u')
        ...
        s = read_cml('--u_exact', '0')
        if s != '0':
            self.u_exact = None
        else:
            self.u_exact = \
                    StringFunction(s, independent_variable='t')
            self.u_exact.vectorize(globals())
        ...

    def rhs(self, u, t):
        """Define the right-hand side in the ODE system."""
        m, f, s, F = \
           self.m, self.friction, self.spring, self.external
        u, dudt = u
        return [dudt,
                (1./m)*(F(t) - f(dudt) - s(u))]
```

The `initialize` method calls `read_cml` from `scitools.misc` to extract the value proceeding the option `-m`. We could have used the `getopt` module, but we aim at reading data from the command line in separate phases in the various classes, and `getopt` does not allow reading the command line more than once. Therefore, we have to use a specialized function `read_cml`.

The exemplified call to `read_cml` implies to look for the command-line argument `-m` for $m$ and treat the next command-line argument as the value of $m$. If the option `-m` is not found at the command line, we use the second argument in the call (here `self.m`) as default value, but returned as a string (since values at the command line are strings).

The class `Solver` follows the design of class `Problem`, but it also has a `solve` method that solves the problem and stores the solution `u` of the ODEs and the time points `t` as attributes:

```
class Solver:
    def initialize(self):
        self.T = eval(read_cml('--T', 4*pi))
        self.dt = eval(read_cml('--dt', pi/20))
        self.method = eval(read_cml('--method', 'RungeKutta4'))

    def solve(self, problem):
        self.solver = self.method(problem.rhs, self.dt)
        ic = [problem.initial_u, problem.initial_dudt]
        self.solver.set_initial_condition(ic, 0.0)
        self.u, self.t = self.solver.solve(self.T)
```

The use of `eval` to initialize `self.T` and `self.dt` allows us to specify these parameters by arithmetic expressions like `pi/40`. Using `eval` on the string specifying the numerical method turns this string into a class type (i.e., a name `'ForwardEuler'` is turned into the class `ForwardEuler` and stored in `self.method`).

The `Visualizer` class holds references to a `Problem` and `Solver` instance and creates plots. The user can specify plots in an interactive dialog in the terminal window. Inside a loop, the user is repeatedly asked to specify a plot until the user responds with `quit`. The specification of a plot can be one of the words `u`, `dudt`, `dudt-u`, `K`, and `wavelength` which means a plot of $u(t)$ versus $t$, $\dot{u}(t)$ versus $t$, $\dot{u}$ versus $u$, $K$ versus $t$, and $u$'s wavelength versus its indices, respectively. The wavelength can be computed from the local maxima of $u$ as explained in Exercise 9.48).

A sketch of class `Visualizer` is given next:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem = problem
        self.solver = solver

    def visualize(self):
        t = self.solver.t   # short form
        u, dudt = self.solver.u[:,0], self.solver.u[:,1]

        # tag all plots with numerical and physical input values:
        title = 'solver=%s, dt=%g, m=%g' % \
               (self.solver.method, self.solver.dt, self.problem.m)
        # can easily get the formula for friction, spring and force
        # if these are string formulas:
        if isinstance(self.problem.friction, StringFunction):
            title += ' f=%s' % str(self.problem.friction)
        if isinstance(self.problem.spring, StringFunction):
            title += ' s=%s' % str(self.problem.spring)
        if isinstance(self.problem.external, StringFunction):
            title += ' F=%s' % str(self.problem.external)

        plot_type = ''
        while plot_type != 'quit':
            plot_type = raw_input('Specify a plot: ')
            figure()
            if plot_type == 'u':
                # plot u vs t
                if self.problem.u_exact is not None:
                    hold('on')
                    # plot self.problem.u_exact vs t
                show()
                hardcopy('tmp_u.eps')
            elif plot_type == 'dudt':
            ...
```

Make a complete implementation of the three proposed classes. Also make a `main` function that (i) creates a problem, solver, and visualizer, (ii) initializes the problem and solver with input data from the command line, (iii) calls the solver, and (iv) calls the visualizer's `visualize` method to create plots. Collect the classes and functions in a module

`oscillator`, which has a call to `main` in the test block. The first task from Exercises 9.46 or 9.47 can now be run as

Terminal

```
oscillator.py --method ForwardEuler --u_exact "cos(t)" \
              --dt "pi/20" --T "5*pi"
```

The other tasks from Exercises 9.46 or 9.47 can be tested similarly.

Explore some of the possibilities of specifying several functions on the command line:

Terminal

```
oscillator.py --method RungeKutta4 --friction "0.1*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "40*pi" --m 10

oscillator.py --method RungeKutta4 --friction "0.8*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "120*pi" --m 50
```

We remark that this module has the same physical and numerical functionality as the module in Exercise 9.47. The only difference is that the code in the modules is organized differently. The organization in terms of classes in the present module avoids shuffling lots of arguments to functions and is often viewed as superior. When solving more complicated problems that result in much larger codes, the class version is usually much simpler to maintain and extend. The reason is that variables are packed together in a few units along with the functionality that operates on the variables.

We also remark that it can be difficult to get `pyreport` to work properly with the present module and further use of it in the forthcoming exercises.

Name of program file: `oscillator.py`. ◇

**Exercise 9.50.** *Allow flexible choice of functions in Exer. 9.49.*

Some typical choices of $f(\dot{u})$, $s(u)$, and $F(t)$ in (9.54) are listed below:

1. Linear friction force (low velocities): $f(\dot{u}) = 6\pi\mu R\dot{u}$ (Stokes drag), where $R$ is the radius of a spherical approximation to the body's geometry, and $\mu$ is the viscosity of the surrounding fluid.
2. Quadratic friction force (high velocities): $f(\dot{u}) = \frac{1}{2}C_D\varrho A|\dot{u}|\dot{u}$, see Exercise 1.10 for explanation of symbols.
3. Linear spring force: $s(u) = ku$, where $k$ is a spring constant.
4. Sinusoidal spring force: $s(u) = k\sin u$, where $k$ is a constant.
5. Cubic spring force: $s(u) = k(u - \frac{1}{6}u^3)$, where $k$ is a spring constant.
6. Sinusoidal external force: $F(t) = F_0 + A\sin\omega t$, where $F_0$ is the mean value of the force, $A$ is the amplitude, and $\omega$ is the frequency.
7. "Bump" force: $F(t) = H(t-t_1)(1-H(t-t_2)F_0$, where $H(t)$ is the Heaviside function from Exercise 2.36, $t_1$ and $t_2$ are two given time

points, and $F_0$ is the size of the force. This $F(t)$ is zero for $t < t_1$ and $t > t_2$, and $F_0$ for $t \in [t_1, t_2]$.

8. Random force 1: $F(t) = F_0 + A \cdot U(t; B)$, where $F_0$ and $A$ are constants, and $U(t; B)$ denotes a function whose value at time $t$ is random and uniformly distributed in the interval $[-B, B]$.
9. Random force 2: $F(t) = F_0 + A \cdot N(t; \mu, \sigma)$, where $F_0$ and $A$ are constants, and $N(t; \mu, \sigma)$ denotes a function whose value at time $t$ is random, Gaussian distributed number with mean $\mu$ and standard deviation $\sigma$.

Make a module `functions` where each of the choices above are implemented as a class with a `__call__` special method. Also add a class `Zero` for a function whose value is always zero. It is natural that the parameters in a function are set as arguments to the constructor. The different classes for spring functions can all have a common base class holding the $k$ parameter as attribute. Name of program file: `functions.py`. ◇

**Exercise 9.51.** *Use the modules from Exer. 9.49 and 9.50.*

The purpose of this exercise is to demonstrate the use of the classes from Exercise 9.50 to solve problems described by (9.54).

With a lot of models for $f(\dot{u})$, $s(u)$, and $F(t)$ available as classes in `functions.py`, it is more challenging to read information about these mathematical functions from the command line[18]. We therefore propose to create the relevant instances in the program and assign them directly to attributes in the `Problem` instance, e.g.,

```
problem = Problem()
problem.m = 1.0
k = 1.2
problem.spring = CubicSpring(k)
...
```

The solver and visualizer objects can still be initialized from the command line, if desired.

Make a separate file, say `oscillator_test.py`, where you import class `Problem`, `Sover`, and `Visualizer`, plus all classes from the `functions` module. Provide a `main1` function with initializations of class `Problem` attributes as indicated above for the case with a Forward Euler method, $m = 1$, $u(0) = 1$, $\dot{u}(0) = 0$, no friction (use class `Zero`), no external forcing (class `Zero`), a linear spring $s(u) = u$, $\Delta t = \pi/20$, $T = 8\pi$, and exact $u(t) = \cos(t)$.

Then make another function `main2` for the case with a 4th-order Runge-Kutta method, $m = 5$, $u(0) = 1$, $\dot{u}(0) = 0$, linear friction $f(\dot{u}) = 0.1\dot{u}$, $s(u) = u$, $F(t) = \sin(\frac{1}{2}t)$, $\Delta t = \pi/80$, $T = 60\pi$, and no knowledge of an exact solution. Let the test block use the first command-line argument to indicate a call to `main1` or `main2`. Name of program file: `oscillator_test.py`. ◇

[18] It can be easily done, however, using `read_cml_func` from `scitools.misc`.

**Exercise 9.52.** *Use the modules from Exer. 9.49 and 9.50.*

Make a program `oscillator_conv.py` where you import the `Problem` and `Solver` classes from the `oscillator` module in Exercise 9.50 and implement a loop in which $\Delta t$ is reduced. The end result should be a plot with the curves $u$ versus $t$ corresponding to the various $\Delta t$ values. Typically, we want to do something like

```
from oscillator import Problem, Solver
from scitools.std import plot, hold, hardcopy

problem = Problem()
problem.initialize()
solver = Solver()
solver.initialize()

# see how the solution changes by halving dt n times:
n = 4
for k in range(n):
    solver.solve(problem)
    u, t = solver.u[:,0], solver.t
    # plot u
    solver.dt = solver.dt/2.0
```

Extend this program with another loop over increasing $m$ values.

Hopefully, you will realize how flexible the classes from Exercises 9.49 and 9.50 are for solving a variety of problems. We can give a set of physical and numerical parameters in a flexible way on the command line, and in the program we may make loops or other constructions to manipulate the input data further.

Name of program file: `oscillator_conv.py`. ◇

# Discrete Calculus

# A

**This appendix is authored by Aslak Tveito**

In this chapter we will discuss how to differentiate and integrate functions on a computer. To do that, we have to care about how to treat mathematical functions on a computer. Handling mathematical functions on computers is not entirely straightforward: A function $f(x)$ contains and infinite amount of information (function values at an infinite number of $x$ values on an interval), while the computer can only store a finite[1] amount of data. Think about the $\cos x$ function. There are typically two ways we can work with this function on a computer. One way is to run an algorithm, like that in Exercise 2.38 on page 108, or we simply call `math.cos(x)` (which runs a similar type of algorithm), to compute an approximation to $\cos x$ for a given $x$, using a finite number of calculations. The other way is to store $\cos x$ values in a table for a finite number of $x$ values[2] and use the table in a smart way to compute $\cos x$ values. This latter way, known as a *discrete* representation of a function, is in focus in the present chapter. With a discrete function representation, we can easily integrate and differentiate the function too. Read on to see how we can do that.

The folder `src/discalc` contains all the program example files referred to in this chapter.

## A.1 Discrete Functions

Physical quantities, such as temperature, density, and velocity, are usually defined as continuous functions of space and time. However, as

[1] Allow yourself a moment or two to think about the terms "finite" and "infinite"; inifinity is not an easy term, but it is not infinitely difficult. Or is it?

[2] Of course, we need to run an algorithm to populate the table with $\cos x$ numbers.

mentioned in above, discrete versions of the functions are more convenient on computers. We will illustrate the concept of discrete functions throught some introductory examples. In fact, we used discrete functions in Chapter 4 to plot curves: We defined a finite set of coordinates `x` and stored the corresponding function values `f(x)` in an array. A plotting program would then draw straight lines between the function values. A discrete representation of a continuous function is, from a programming point of view, nothing but storing a finite set of coordinates and function values in an array. Nevertheless, we will in this chapter be more formal and describe discrete functions by precise mathematical terms.

### A.1.1 The Sine Function

Suppose we want to generate a plot of the sine function for values of $x$ between 0 and $\pi$. To this end, we define a set of $x$-values and an associated set of values of the sine function. More precisely, we define $n+1$ points by

$$x_i = ih \text{ for } i = 0, 1, \ldots, n \tag{A.1}$$

where $h = \pi/n$ and $n \geqslant 1$ is an integer. The associated function values are defined as

$$s_i = \sin(x_i) \text{ for } i = 0, 1, \ldots, n. \tag{A.2}$$

Mathematically, we have a sequence of coordinates $(x_i)_{i=0}^n$ and of function values $(s_i)_{i=0}^n$ (see the start of Chapter 5 for an explanation of the notation and the sequence concept). Often we "merge" the two sequences to one sequence of points: $(x_i, s_i)_{i=0}^n$. Sometimes we also use a shorter notation, just $x_i$, $s_i$, or $(x_i, s_i)$ if the exact limits are not of importance. The set of coordinates $(x_i)_{i=0}^n$ constitutes a *mesh* or a *grid*. The individual coordinates $x_i$ are known as *nodes* in the mesh (or grid). The discrete representation of the sine function on $[0, \pi]$ consists of the mesh and the corresponding sequence of function values $(s_i)_{i=0}^n$ at the nodes. The parameter $n$ is often referred to as the *mesh resolution*.

In a program, we represent the mesh by a coordinate array, say `x`, and the function values by another array, say `s`. To plot the sine function we can simply write

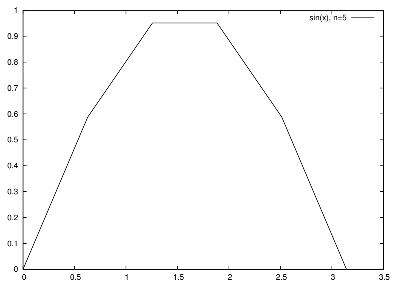
```
from scitools.std import *

n = int(sys.argv[1])

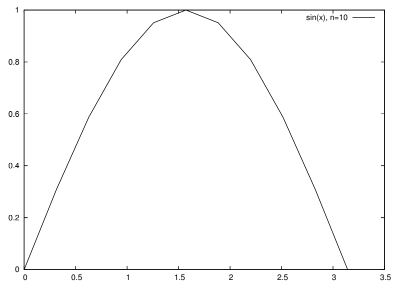
x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```

Figure A.1 shows the resulting plot for $n = 5, 10, 20$ and 100. As pointed out in Chapter 4, the curve looks smoother the more points

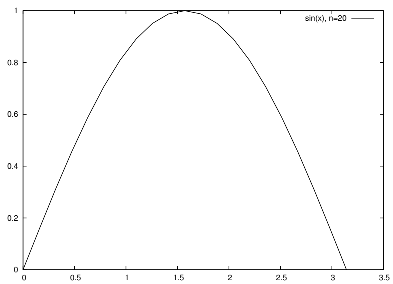
we use, and since $\sin(x)$ is a smooth function, the plots in Figures A.1a and A.1b do not look sufficiently good. However, we can with our eyes hardly distinguish the plot with 100 points from the one with 20 points, so 20 points seem sufficient in this example.



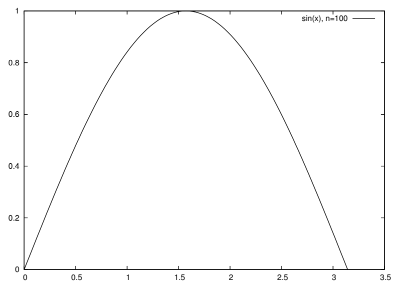

(a) $n = 5$




(b) $n = 10$




(c) $n = 20$




(d) $n = 100$

**Fig. A.1** Plots of $\sin(x)$ with various $n$.

There are no tests on the validity of the input data (`n`) in the previous program. A program including these tests reads[3]:

```
#!/usr/bin/env python
from scitools.std import *

try:
    n = int(sys.argv[1])
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```

Such tests are important parts of a good programming philosophy. However, for the programs displayed in this and the next chapter, we skip such tests in order to make the programs more compact and readable as part of the rest of the text and to enable focus on the mathematics in the programs. In the versions of these programs in the

[3] For an explanation of the first line of this program, see Appendix E.1

files that can be downloaded you will, hopefully, always find a test on input data.

### A.1.2 Interpolation

Suppose we have a discrete representation of the sine function: $(x_i, s_i)_{i=0}^n$. At the nodes we have the exact sine values $s_i$, but what about the points in between these nodes? Finding function values between the nodes is called *interpolation*, or we can say that we *interpolate* a discrete function.

A graphical interpolation procedure could be to look at one of the plots in Figure A.1 to find the function value corresponding to a point $x$ between the nodes. Since the plot is a straight line from node value to node value, this means that a function value between two nodes is found from a straight line approximation[4] to the underlying continuous function. We formulate this procedure precisely in terms of mathematics in the next paragraph.

Assume that we know that a given $x^*$ lies in the interval from $x = x_k$ to $x_{k+1}$, where the integer $k$ is given. In the interval $x_k \leqslant x < x_{k+1}$, we define the linear function that passes through $(x_k, s_k)$ and $(x_{k+1}, s_{k+1})$:

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k). \tag{A.3}$$

That is, $S_k(x)$ coincides with $\sin(x)$ at $x_k$ and $x_{k+1}$, and between these nodes, $S_k(x)$ is linear. We say that $S_k(x)$ interpolates the discrete function $(x_i, s_i)_{i=0}^n$ on the interval $[x_k, x_{k+1}]$.

### A.1.3 Evaluating the Approximation

Given the values $(x_i, s_i)_{i=0}^n$ and the formula $(A.3)$, we want to compute an approximation of the sine function for any $x$ in the interval from $x = 0$ to $x = \pi$. In order to do that, we have to compute $k$ for a given value of $x$. More precisely, for a given $x$ we have to find k such that $x_k \leqslant x \leqslant x_{k+1}$. We can do that by defining

$$k = \lfloor x/h \rfloor$$

where the function $\lfloor z \rfloor$ denotes the largest integer that is smaller than $z$. In Python, $\lfloor z \rfloor$ is computed by `int(z)`. The program below takes $x$ and $n$ as input and computes the approximation of $\sin(x)$. The program

[4] Strictly speaking, we also assume that the function to be interpolated is rather smooth. It is easy to see that if the function is very wild, i.e., the values of the function changes very rapidly, this procedure may fail even for very large values of $n$. Chapter 4.4.2 provides an example.

prints the approximation $S(x)$ and the exact[5] value of $\sin(x)$ so we can look at the development of the error when $n$ is increased.

```
from numpy import *
import sys

xp = eval(sys.argv[1])
n = int(sys.argv[2])

def S_k(k):
    return s[k] + \
           ((s[k+1] - s[k])/(x[k+1] - x[k]))*(xp - x[k])
h = pi/n
x = linspace(0, pi, n+1)
s = sin(x)
k = int(xp/h)

print 'Approximation of sin(%s):  ' % xp, S_k(k)
print 'Exact value of sin(%s):    ' % xp, sin(xp)
print 'Eror in approximation:     ', sin(xp) - S_k(k)
```

To study the approximation, we put $x = \sqrt{2}$ and use the program `eval_sine.py` for $n = 5, 10$ and 20.

Terminal

```
eval_sine.py 'sqrt(2)' 5
Approximation of sin(1.41421356237):   0.951056516295
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      0.0367094296976
```

Terminal

```
eval_sine.py 'sqrt(2)' 10
Approximation of sin(1.41421356237):   0.975605666221
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      0.0121602797718
```

Terminal

```
eval_sine.py 'sqrt(2)' 20
Approximation of sin(1.41421356237):   0.987727284363
Exact value of sin(1.41421356237):     0.987765945993
Eror in approximation:      3.86616296923e-05
```

Note that the error is reduced as the $n$ increases.

### A.1.4 Generalization

In general, we can create a discrete version of a continuous function as follows. Suppose a continuous function $f(x)$ is defined on an interval

[5] The value is not really exact – it is the value of $\sin(x)$ provided by the computer, `math.sin(x)`, and this value is calculated from an algorithm that only yields an approximation to $\sin(x)$. Exercise 2.38 provides an example of the type of algorithm in question.

ranging from $x = a$ to $x = b$, and let $n \geqslant 1$, be a given integer. Define the distance between nodes,

$$h = \frac{b-a}{n},$$

and the nodes

$$x_i = a + ih \text{ for } i = 0, 1, \ldots, n. \tag{A.4}$$

The discrete function values are given by

$$y_i = f(x_i) \text{ for } i = 0, 1, \ldots, n. \tag{A.5}$$

Now, $(x_i, y_i)_{i=0}^n$ is the discrete version of the continuous function $f(x)$. The program `discrete_func.py` takes $f, a, b$ and $n$ as input, computes the discrete version of $f$, and then applies the discrete version to make a plot of $f$.

```
def discrete_func(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    for i in xrange(len(x)):
        y[i] = func(x[i])
    return x, y

from scitools.std import *

f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])
f = StringFunction(f_formula)

x, y = discrete_func(f, a, b, n)
plot(x, y)
```

We can equally well make a vectorized version of the `discrete_func` function:

```
def discrete_func(f, a, b, n):
    x = linspace(a, b, n+1)
    y = f(x)
    return x, y
```

However, for the `StringFunction` tool to work properly in vectorized mode, we need to follow the recipe in Chapter 4.4.3:

```
f = StringFunction(f_formula)
f.vectorize(globals())
```

The corresponding vectorized program is found in the file `discrete_func_vec.py`.

## A.2 Differentiation Becomes Finite Differences

You have heard about derivatives. Probably, the following formulas are well known to you:

$$\frac{d}{dx}\sin(x) = \cos(x),$$
$$\frac{d}{dx}\ln(x) = \frac{1}{x},$$
$$\frac{d}{dx}x^m = mx^{m-1},$$

But why is differentiation so important? The reason is quite simple: The derivative is a mathematical expression of change. And change is, of course, essential in modeling various phenomena. If we know the state of a system, and we know the laws of change, then we can, in principle, compute the future of that system. Appendix B treats this topic in detail. Chapter 5 also computes the future of systems, based on modeling changes, but without using differentiation. In Appendix B you will see that reducing the step size in the difference equations in Chapter 5 results in derivatives instead of pure differences. However, differentiation of continuous functions is somewhat hard on a computer, so we often end up replacing the derivatives by differences. This idea is quite general, and every time we use a discrete representation of a function, differentiation becomes differences, or *finite differences* as we usually say.

The mathematical definition of differentiation reads

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}\,.$$

You have probably seen this definition many times, but have you understood what it means and do you think the formula has a great practical value? Although the definition requires that we pass to the limit, we obtain quite good approximations of the derivative by using a fixed positive value of $\varepsilon$. More precisely, for a small $\varepsilon > 0$, we have

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}\,.$$

The fraction on the right-hand side is a finite difference approximation to the derivative of $f$ at the point $x$. Instead of using $\varepsilon$ it is more common to introduce $h = \varepsilon$ in finite differences, i.e., we like to write

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}\,. \qquad \text{(A.6)}$$

### A.2.1 Differentiating the Sine Function

In order to get a feeling for how good the approximation (A.6) to the derivative really is, we explore an example. Consider $f(x) = \sin(x)$ and the associated derivative $f'(x) = \cos(x)$. If we put $x = 1$,we have

$$f'(1) = \cos(1) \approx 0.540\,,$$

and by putting $h = 1/100$ in (A.6) we get

$$f'(1) \approx \frac{f(1 + 1/100) - f(1)}{1/100} = \frac{\sin(1.01) - \sin(1)}{0.01} \approx 0.536\,.$$

The program `forward_diff.py`, shown below, computes the derivative of $f(x)$ using the approximation (A.6), where $x$ and $h$ are input parameters.

```
def diff(f, x, h):
    return (f(x+h) - f(x))/float(h)

from math import *
import sys

x = eval(sys.argv[1])
h = eval(sys.argv[2])

approx_deriv = diff(sin, x, h)
exact = cos(x)
print 'The approximated value is: ', approx_deriv
print 'The correct value is:      ', exact
print 'The error is:              ', exact - approx_deriv
```

Running the program for $x = 1$ and $h = 1/1000$ gives

Terminal

```
forward_diff.py 1 0.001
The approximated value is:  0.53988148036
The correct value is:       0.540302305868
The error is:               0.000420825507813
```

### A.2.2 Differences on a Mesh

Frequently, we will need finite difference approximations to a discrete function defined on a mesh. Suppose we have a discrete representation of the sine function: $(x_i, s_i)_{i=0}^{n}$, as introduced in Chapter A.1.1. We want to use (A.6) to compute approximations to the derivative of the sine function at the nodes in the mesh. Since we only have function values at the nodes, the $h$ in (A.6) must be the difference between nodes, i.e., $h = x_{i+1} - x_i$. At node $x_i$ we then have the following approximation of the derivative:

$$z_i = \frac{s_{i+1} - s_i}{h}, \tag{A.7}$$

for $i = 0, 1, \ldots, n-1$. Note that we have not defined an approximate derivative at the end point $x = x_n$. We cannot apply (A.7) directly since $s_{n+1}$ is undefined (outside the mesh). However, the derivative of a function can also be defined as

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x) - f(x-\varepsilon)}{\varepsilon},$$

which motivates the following approximation for a given $h > 0$,

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}\,. \tag{A.8}$$

This alternative approximation to the derivative is referred to as a *backward difference* formula, whereas the expression (A.6) is known as a *forward difference* formula. The names are natural: The forward formula goes forward, i.e., in the direction of increasing $x$ and $i$ to collect information about the change of the function, while the backward formula goes backwards, i.e., toward smaller $x$ and $i$ value to fetch function information.

At the end point we can apply the backward formula and thus define

$$z_n = \frac{s_n - s_{n-1}}{h}\,. \tag{A.9}$$

We now have an approximation to the derivative at all the nodes. A plain specialized program for computing the derivative of the sine function on a mesh and comparing this discrete derivative with the exact derivative is displayed below (the name of the file is `diff_sine_plot1.py`).

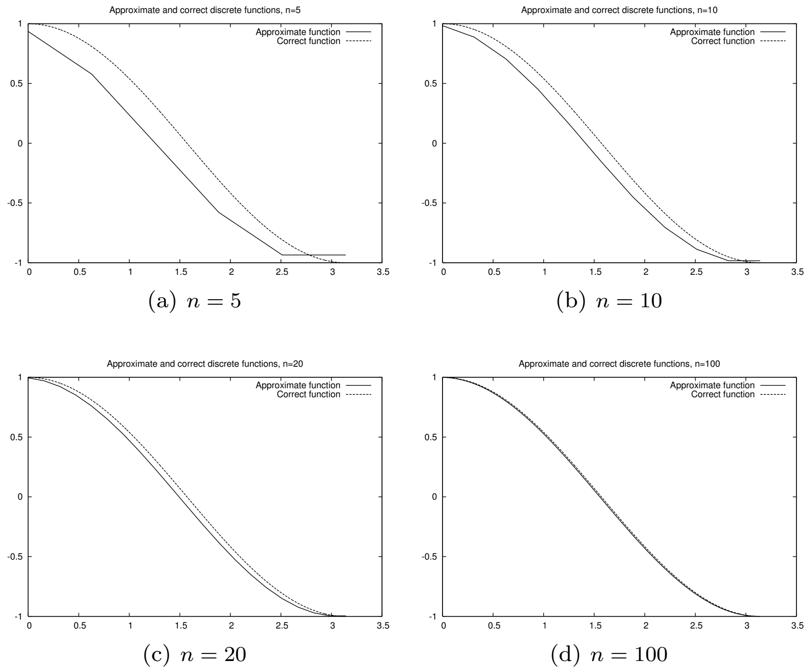
```
from scitools.std import *

n = int(sys.argv[1])

h = pi/n
x = linspace(0, pi, n+1)
s = sin(x)
z = zeros(len(s))
for i in xrange(len(z)-1):
    z[i] = (s[i+1] - s[i])/h
# special formula for end point_
z[-1] = (s[-1] - s[-2])/h
plot(x, z)

xfine = linspace(0, pi, 1001) # for more accurate plot
exact = cos(xfine)
hold()
plot(xfine, exact)
legend('Approximate function', 'Correct function')
title('Approximate and discrete functions, n=%d' % n)
```

In Figure A.2 we see the resulting graphs for $n = 5, 10, 20$ and 100. Again, we note that the error is reduced as $n$ increases.




(a) $n = 5$ (b) $n = 10$ (c) $n = 20$ (d) $n = 100$

**Fig. A.2** Plots for exact and approximate derivatives of $\sin(x)$ with varying values of the resolution $n$.

### A.2.3 Generalization

The discrete version of a continuous function $f(x)$ defined on an interval $[a, b]$ is given by $(x_i, y_i)_{i=0}^n$ where

$$x_i = a + ih,$$

and

$$y_i = f(x_i)$$

for $i = 0, 1, \ldots, n$. Here, $n \geqslant 1$ is a given integer, and the spacing between the nodes is given by

$$h = \frac{b-a}{n}.$$

A discrete approximation of the derivative of $f$ is given by $(x_i, z_i)_{i=0}^n$ where

$$z_i = \frac{y_{i+1} - y_i}{h}$$

$i = 0, 1, \ldots, n-1$, and

$$z_n = \frac{y_n - y_{n-1}}{h}.$$

The collection $(x_i, z_i)_{i=0}^n$ is the discrete derivative of the discrete version $(x_i, f_i)_{i=0}^n$ of the continuous function $f(x)$. The program below, found in the file `diff_func.py`, takes $f, a, b$ and $n$ as input and computes the discrete derivative of $f$ on the mesh implied by $a$, $b$, and $h$, and then a plot of $f$ and the discrete derivative is made.

```
def diff(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    z = zeros(len(x))
    h = (b-a)/float(n)
    for i in xrange(len(x)):
        y[i] = func(x[i])
    for i in xrange(len(x)-1):
        z[i] = (y[i+1] - y[i])/h
    z[n] = (y[n] - y[n-1])/h
    return y, z

from scitools.std import *
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])

f = StringFunction(f_formula)
y, z = diff(f, a, b, n)
plot(x, y, 'r-', x, z, 'b-',
     legend=('function', 'derivative'))
```

## A.3 Integration Becomes Summation

Some functions can be integrated analytically. You may remember[6] the following cases,

$$\int x^m dx = \frac{1}{m+1}x^{m+1} \text{ for } m \neq -1,$$
$$\int \sin(x)dx = -\cos(x),$$
$$\int \frac{x}{1+x^2}dx = \frac{1}{2}\ln\left(x^2+1\right).$$

These are examples of so-called indefinite integrals. If the function can be integrated analytically, it is straightforward to evaluate an associated definite integral. For example, we have[7]

[6] Actually, we congratulate you if you remember the third one!

[7] Recall, in general, that

$$[f(x)]_a^b = f(b) - f(a).$$

$$\int_0^1 x^m dx = \left[\frac{1}{m+1}x^{m+1}\right]_0^1 = \frac{1}{m+1},$$
$$\int_0^\pi \sin(x)dx = [-\cos(x)]_0^\pi = 2,$$
$$\int_0^1 \frac{x}{1+x^2}dx = \left[\frac{1}{2}\ln\left(x^2+1\right)\right]_0^1 = \frac{1}{2}\ln 2.$$

But lots of functions cannot be integrated analytically and therefore definite integrals must be computed using some sort of numerical approximation. Above, we introduced the discrete version of a function, and we will now use this construction to compute an approximation of a definite integral.

### A.3.1 Dividing into Subintervals

Let us start by considering the problem of computing the integral of $\sin(x)$ from $x = 0$ to $x = \pi$. This is not the most exciting or challenging mathematical problem you can think of, but it is good practice to start with a problem you know well when you want to learn a new method. In Chapter A.1.1 we introduce a discrete function $(x_i, s_i)_{i=0}^n$ where $h = \pi/n, s_i = \sin(x_i)$ and $x_i = ih$ for $i = 0, 1, \ldots, n$. Furthermore, in the interval $x_k \leqslant x < x_{k+1}$, we defined the linear function

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k).$$

We want to compute an approximation of the integral of the function $\sin(x)$ from $x = 0$ to $x = \pi$. The integral

$$\int_0^\pi \sin(x)dx$$

can be divided into subintegrals defined on the intervals $x_k \leqslant x < x_{k+1}$, leading to the following sum of integrals:

$$\int_0^\pi \sin(x)dx = \sum_{k=0}^{n-1}\int_{x_k}^{x_{k+1}} \sin(x)dx\,.$$

To get a feeling for this split of the integral, let us spell the sum out in the case of only four subintervals. Then $n = 4$, $h = \pi/4$,

$$
\begin{aligned}
x_0 &= 0,\\
x_1 &= \pi/4,\\
x_2 &= \pi/2,\\
x_3 &= 3\pi/4\\
x_4 &= \pi.
\end{aligned}
$$

The interval from 0 to $\pi$ is divided into four intervals of equal length, and we can divide the integral similarily,

$$
\int_0^{\pi} \sin(x)dx = \int_{x_0}^{x_1} \sin(x)dx + \int_{x_1}^{x_2} \sin(x)dx + \int_{x_2}^{x_3} \sin(x)dx + \int_{x_3}^{x_4} \sin(x)dx\,. \tag{A.10}
$$

So far we have changed nothing – the integral can be split in this way – with no approximation at all. But we have reduced the problem of approximating the integral

$$
\int_0^{\pi} \sin(x)dx
$$

down to approximating integrals on the subintervals, i.e. we need approxiamtions of all the following integrals

$$
\int_{x_0}^{x_1} \sin(x)dx,\ \int_{x_1}^{x_2} \sin(x)dx,\ \int_{x_2}^{x_3} \sin(x)dx,\ \int_{x_3}^{x_4} \sin(x)dx\,.
$$

The idea is that the function to be integrated changes less over the subintervals than over the whole domain $[0, \pi]$ and it might be reasonable to approximate the sine by a straight line, $S_k(x)$, over each subinterval. The integration over a subinterval will then be very easy.

### A.3.2 Integration on Subintervals

The task now is to approximate integrals on the form

$$
\int_{x_k}^{x_{k+1}} \sin(x)dx.
$$

Since

$$
\sin(x) \approx S_k(x)
$$

on the interval $(x_k, x_{k+1})$, we have

$$
\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \int_{x_k}^{x_{k+1}} S_k(x)dx.
$$

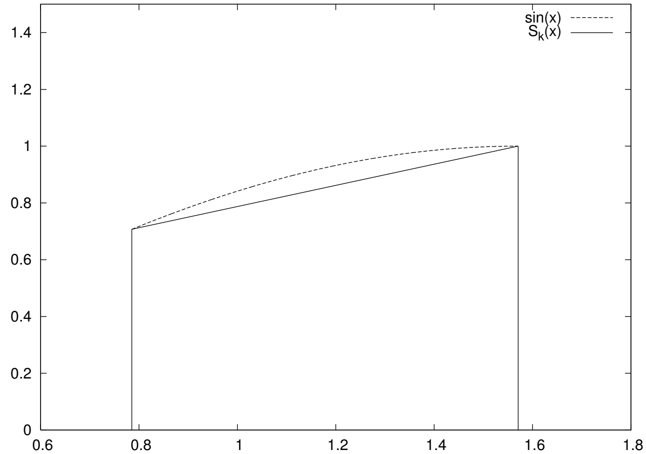



**Fig. A.3** $S_k(x)$ and $\sin(x)$ on the interval $(x_k, x_{k+1})$ for $k = 1$ and $n = 4$.

In Figure A.3 we have graphed $S_k(x)$ and $\sin(x)$ on the interval $(x_k, x_{k+1})$ for $k = 1$ in the case of $n = 4$. We note that the integral of $S_1(x)$ on this interval equals the area of a trapezoid, and thus we have

$$\int_{x_1}^{x_2} S_1(x)dx = \frac{1}{2}\left(S_1(x_2) + S_1(x_1)\right)(x_2 - x_1),$$

so

$$\int_{x_1}^{x_2} S_1(x)dx = \frac{h}{2}\left(s_2 + s_1\right),$$

and in general we have

$$\begin{aligned}\int_{x_k}^{x_{k+1}} \sin(x)dx &\approx \frac{1}{2}\left(s_{k+1} + s_k\right)(x_{k+1} - x_k) \\ &= \frac{h}{2}\left(s_{k+1} + s_k\right).\end{aligned}$$

### A.3.3 Adding the Subintervals

By adding the contributions from each subinterval, we get

$$\begin{aligned}\int_0^{\pi} \sin(x)dx &= \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x)dx \\ &\approx \sum_{k=0}^{n-1} \frac{h}{2}\left(s_{k+1} + s_k\right),\end{aligned}$$

so

$$\int_0^{\pi} \sin(x)dx \approx \frac{h}{2}\sum_{k=0}^{n-1}\left(s_{k+1} + s_k\right). \tag{A.11}$$

In the case of $n = 4$, we have

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2}\left[(s_1 + s_0) + (s_2 + s_1) + (s_3 + s_2) + (s_4 + s_3)\right]$$
$$= \frac{h}{2}\left[s_0 + 2\left(s_{1+}s_{2+}s_3\right) + s_4\right] .$$

One can show that (A.11) can be alternatively expressed as[8]

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2}\left[s_0 + 2\sum_{k=1}^{n-1} s_k + s_n\right] . \tag{A.12}$$

This approximation formula is referred to as the Trapezoidal rule of numerical integration. Using the more general program `trapezoidal.py`, presented in the next section, on integrating $\int_0^\pi \sin(x)dx$ with $n = 5, 10, 20$ and 100 yields the numbers 1.5644, 1.8864, 1.9713, and 1.9998 respectively. These numbers are to be compared to the exact value 2. As usual, the approximation becomes better the more points ($n$) we use.

### A.3.4 Generalization

An approximation of the integral

$$\int_a^b f(x)dx$$

can be computed using the discrete version of a continuous function $f(x)$ defined on an interval $[a, b]$. We recall that the discrete version of $f$ is given by $(x_i, y_i)_{i=0}^n$ where

$$x_i = a + ih, \text{ and } y_i = f(x_i)$$

for $i = 0, 1, \ldots, n$. Here, $n \geqslant 1$ is a given integer and $h = (b - a)/n$. The Trapezoidal rule can now be written as

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right] .$$

The program `trapezoidal.py` implements the Trapezoidal rule for a general function $f$.

```
def trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    I = f(a) + f(b)
    for k in xrange(1, n, 1):
```

[8] There are fewer arithmetic operations associated with (A.12) than with (A.11), so the former will lead to faster code.

```
        x = a + k*h
        I += 2*f(x)
    I *= h/2
    return I


from math import *
from scitools.StringFunction import StringFunction
import sys

def test(argv=sys.argv):
    f_formula = argv[1]
    a = eval(argv[2])
    b = eval(argv[3])
    n = int(argv[4])

    f = StringFunction(f_formula)
    I = trapezoidal(f, a, b, n)
    print 'Approximation of the integral: ', I

if __name__ == '__main__':
    test()
```

We have made the file as module such that you can easily import the `trapezoidal` function in another program. Let us do that: We make a table of how the approximation and the associated error of an integral are reduced as $n$ is increased. For this purpose, we want to integrate $\int_{t_1}^{t_2} g(t)dt$, where

$$g(t) = -ae^{-at}\sin(\pi wt) + \pi we^{-at}\cos(\pi wt)\,.$$

The exact integral $G(t) = \int g(t)dt$ equals

$$G(t) = e^{-at}\sin(\pi wt)\,.$$

Here, $a$ and $w$ are real numbers that we set to $1/2$ and $1$, respectively, in the program. The integration limits are chosen as $t_1 = 0$ and $t_2 = 4$. The integral then equals zero. The program and its output appear below.

```
from trapezoidal import trapezoidal
from math import exp, sin, cos, pi

def g(t):
    return -a*exp(-a*t)*sin(pi*w*t) + pi*w*exp(-a*t)*cos(pi*w*t)

def G(t):  # integral of g(t)
    return exp(-a*t)*sin(pi*w*t)

a = 0.5
w = 1.0
t1 = 0
t2 = 4
exact = G(t2) - G(t1)
for n in 2, 4, 8, 16, 32, 64, 128, 256, 512:
    approx = trapezoidal(g, t1, t2, n)
    print 'n=%3d approximation=%12.5e  error=%12.5e' % \
          (n, approx, exact-approx)
```

```
n=  2 approximation= 5.87822e+00  error=-5.87822e+00
n=  4 approximation= 3.32652e-01  error=-3.32652e-01
n=  8 approximation= 6.15345e-02  error=-6.15345e-02
n= 16 approximation= 1.44376e-02  error=-1.44376e-02
n= 32 approximation= 3.55482e-03  error=-3.55482e-03
n= 64 approximation= 8.85362e-04  error=-8.85362e-04
n=128 approximation= 2.21132e-04  error=-2.21132e-04
n=256 approximation= 5.52701e-05  error=-5.52701e-05
n=512 approximation= 1.38167e-05  error=-1.38167e-05
```

We see that the error is reduced as we increase $n$. In fact, as $n$ is doubled we realize that the error is roughly reduced by a factor of 4, at least when $n > 8$. This is an important property of the Trapezoidal rule, and checking that a program reproduces this property is an important check of the validity of the implementation.

## A.4 Taylor Series

The single most important mathematical tool in computational science is the Taylor series. It is used to derive new methods and also for the analysis of the accuracy of approximations. We will use the series many times in this text. Right here, we just introduce it and present a few applications.

### A.4.1 Approximating Functions Close to One Point

Suppose you know the value of a function $f$ at some point $x_0$, and you are interested in the value of $f$ close to $x$. More precisely, suppose we know $f(x_0)$ and we want an approximation of $f(x_0 + h)$ where $h$ is a small number. If the function is smooth and $h$ is really small, our first approximation reads

$$f(x_0 + h) \approx f(x_0). \tag{A.13}$$

That approximation is, of course, not very accurate. In order to derive a more accurate approximation, we have to know more about $f$ at $x_0$. Suppose that we know the value of $f(x_0)$ and $f'(x_0)$, then we can find a better approximation of $f(x_0 + h)$ by recalling that

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Hence, we have

$$f(x_0 + h) \approx f(x_0) + hf'(x_0). \tag{A.14}$$

### A.4.2 Approximating the Exponential Function

Let us be a bit more specific and consider the case of

$$f(x) = e^x$$

around

$$x_0 = 0.$$

Since $f'(x) = e^x$, we have $f'(0) = 1$, and then it follows from $(A.14)$ that

$$e^h \approx 1 + h.$$

The little program below (found in `taylor1.py`) prints $e^h$ and $1 + h$ for a range of $h$ values.

```
from math import exp
for h in 1, 0.5, 1/20.0, 1/100.0, 1/1000.0:
    print 'h=%8.6f exp(h)=%11.5e  1+h=%g' % (h, exp(h), 1+h)
```

```
h=1.000000 exp(h)=2.71828e+00  1+h=2
h=0.500000 exp(h)=1.64872e+00  1+h=1.5
h=0.050000 exp(h)=1.05127e+00  1+h=1.05
h=0.010000 exp(h)=1.01005e+00  1+h=1.01
h=0.001000 exp(h)=1.00100e+00  1+h=1.001
```

As expected, $1 + h$ is a good approximation to $e^h$ the smaller $h$ is.

### A.4.3 More Accurate Expansions

The approximations given by $(A.13)$ and $(A.14)$ are referred to as Taylor series. You can read much more about Taylor series in any Calculus book. More specifically, (A.13) and (A.14) are known as the zeroth- and first-order Taylor series, respectively. The second-order Taylor series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0), \tag{A.15}$$

the third-order series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0), \tag{A.16}$$

and the fourth-order series reads

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + \frac{h^4}{24} f''''(x_0). \tag{A.17}$$

In general, the $n$-th order Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0), \tag{A.18}$$

where we recall that $f^{(k)}$ denotes the $k-th$ derivative of $f$, and

$$k! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (k-1) \cdot k$$

is the factorial (cf. Exercise 2.33). By again considering $f(x) = e^x$ and $x_0 = 0$, we have

$$f(x_0) = f'(x_0) = f''(x_0) = f'''(x_0) = f''''(x_0) = 1$$

which gives the following Taylor series:

$$\begin{aligned}
e^h &\approx 1, && \text{zeroth-order,}\\
e^h &\approx 1 + h, && \text{first-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2, && \text{second-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3, && \text{third-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + \tfrac{1}{24}h^4, && \text{fourth-order.}
\end{aligned}$$

The program below, called `taylor2.py`, prints the error of these approximations for a given value of $h$ (note that we can easily build up a Taylor series in a list by adding a new term to the last computed term in the list).

```
from math import exp
import sys
h = float(sys.argv[1])

Taylor_series = []
Taylor_series.append(1)
Taylor_series.append(Taylor_series[-1] + h)
Taylor_series.append(Taylor_series[-1] + (1/2.0)*h**2)
Taylor_series.append(Taylor_series[-1] + (1/6.0)*h**3)
Taylor_series.append(Taylor_series[-1] + (1/24.0)*h**4)

print 'h =', h
for order in range(len(Taylor_series)):
    print 'order=%d, error=%g' % \
          (order, exp(h) - Taylor_series[order])
```

By running the program with $h = 0.2$, we have the following output:

```
h = 0.2
order=0, error=0.221403
order=1, error=0.0214028
order=2, error=0.00140276
order=3, error=6.94248e-05
order=4, error=2.75816e-06
```

We see how much the approximation is improved by adding more terms. For $h = 3$ all these approximations are useless:

```
h = 3.0
order=0, error=19.0855
order=1, error=16.0855
order=2, error=11.5855
order=3, error=7.08554
order=4, error=3.71054
```

However, by adding more terms we can get accurate results for any $h$. The method from Chapter 5.1.7 computes the Taylor series for $e^x$ with $n$ terms in general. Running the associated program `exp_Taylor_series_diffeq.py` for various values of $h$ shows how much is gained by adding more terms to the Taylor series. For $h = 3$,

$e^3 = 20.086$ and we have

| $n+1$ | Taylor series |
|---|---|
| 2 | 4 |
| 4 | 13 |
| 8 | 19.846 |
| 16 | 20.086 |

For $h = 50$, $e^{50} = 5.1847 \cdot 10^{21}$ and we have

| $n+1$ | Taylor series |
|---|---|
| 2 | 51 |
| 4 | $2.2134 \cdot 10^4$ |
| 8 | $1.7960 \cdot 10^8$ |
| 16 | $3.2964 \cdot 10^{13}$ |
| 32 | $1.3928 \cdot 10^{19}$ |
| 64 | $5.0196 \cdot 10^{21}$ |
| 128 | $5.1847 \cdot 10^{21}$ |

Here, the evolution of the series as more terms are added is quite dramatic (and impressive!).

### A.4.4 Accuracy of the Approximation

Recall that the Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0). \tag{A.19}$$

This can be rewritten as an equality by introducing an error term,

$$f(x_0 + h) = \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0) + O(h^{n+1}). \tag{A.20}$$

Let's look a bit closer at this for $f(x) = e^x$. In the case of $n = 1$, we have

$$e^h = 1 + h + O(h^2). \tag{A.21}$$

This means that there is a constant $c$ that does not depend on $h$ such that

$$\left| e^h - (1 + h) \right| \leqslant ch^2, \tag{A.22}$$

so the error is reduced quadratically in $h$. This means that if we compute the fraction

$$q_h^1 = \frac{\left| e^h - (1 + h) \right|}{h^2},$$

we expect it to be bounded as $h$ is reduced. The program `taylor_err1.py` prints $q_h^1$ for $h = 1/10, 1/20, 1/100$ and $1/1000$.

```
from numpy import exp, abs

def q_h(h):
    return abs(exp(h) - (1+h))/h**2
```

```
print "  h     q_h"
for h in 0.1, 0.05, 0.01, 0.001:
    print "%5.3f %f" %(h, q_h(h))
```

We can run the program and watch the output:

Terminal

```
taylor_err1.py
  h     q_h
0.100 0.517092
0.050 0.508439
0.010 0.501671
0.001 0.500167
```

We observe that $q_h \approx 1/2$ and it is definitely bounded independent of $h$. We can now rewrite all the approximations of $e^h$ defined above in term of equalities:

$$
\begin{aligned}
e^h &= 1 + O(h), && \text{zeroth-order,}\\
e^h &= 1 + h + O(h^2), && \text{first-order,}\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + O(h^3), && \text{second-order,}\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + O(h^4), && \text{third-order,}\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + \tfrac{1}{24}h^4 + O(h^5), && \text{fourth-order.}
\end{aligned}
$$

The program `taylor_err2.py` prints

$$
\begin{aligned}
q_h^0 &= \frac{\left|e^h - 1\right|}{h},\\
q_h^1 &= \frac{\left|e^h - (1+h)\right|}{h^2},\\
q_h^2 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2}\right)\right|}{h^3},\\
q_h^3 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2} + \frac{h^3}{6}\right)\right|}{h^4},\\
q_h^4 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24}\right)\right|}{h^5},
\end{aligned}
$$

for $h = 1/5, 1/10, 1/20$ and $1/100$.

```
from numpy import exp, abs

def q_0(h):
    return abs(exp(h) - 1) / h
def q_1(h):
    return abs(exp(h) - (1 + h)) / h**2
def q_2(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2)) / h**3
def q_3(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + \
                         (1/6.0)*h**3)) / h**4
def q_4(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + (1/6.0)*h**3 + \
```

```
                         (1/24.0)*h**4)) / h**5
hlist = [0.2, 0.1, 0.05, 0.01]
print "%-05s %-09s %-09s %-09s %-09s %-09s" \
      %("h", "q_0", "q_1", "q_2", "q_3", "q_4")
for h in hlist:
    print "%.02f  %04f  %04f  %04f  %04f  %04f" \
          %(h, q_0(h), q_1(h), q_2(h), q_3(h), q_4(h))
```

By using the program, we get the following table:

```
h     q_0       q_1       q_2       q_3       q_4
0.20  1.107014  0.535069  0.175345  0.043391  0.008619
0.10  1.051709  0.517092  0.170918  0.042514  0.008474
0.05  1.025422  0.508439  0.168771  0.042087  0.008403
0.01  1.005017  0.501671  0.167084  0.041750  0.008344
```

Again we observe that the error of the approximation behaves as indicated in $(A.20)$.

### A.4.5 Derivatives Revisited

We observed aboved that

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}.$$

By using the Taylor series, we can obtain this approximation directly, and also get an indication of the error of the approximation. From $(A.20)$ it follows that

$$f(x+h) = f(x) + hf'(x) + O(h^2),$$

and thus

$$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h), \tag{A.23}$$

so the error is proportional to $h$. We can investigate if this is the case through some computer experiments. Take $f(x) = \ln(x)$, so that $f'(x) = 1/x$. The program `diff_ln_err.py` prints $h$ and

$$\frac{1}{h}\left|f'(x) - \frac{f(x+h)-f(x)}{h}\right| \tag{A.24}$$

at $x = 10$ for a range of $h$ values.

```
def error(h):
    return (1.0/h)*abs(df(x) - (f(x+h)-f(x))/h)

from math import log as ln

def f(x):
   return ln(x)

def df(x):
    return 1.0/x

x = 10
hlist = []
```

```
for h in 0.2, 0.1, 0.05, 0.01, 0.001:
    print "%.4f   %4f" % (h, error(h))
```

From the output

```
0.2000   0.004934
0.1000   0.004967
0.0500   0.004983
0.0100   0.004997
0.0010   0.005000
```

we observe that the quantity in (A.24) is constant ($\approx 0.5$) independent of $h$, which indicates that the error is proportional to $h$.

### A.4.6 More Accurate Difference Approximations

We can also use the Taylor series to derive more accurate approximations of the derivatives. From $(A.20)$, we have

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \tag{A.25}$$

By using $-h$ insted of $h$, we get

$$f(x-h) \approx f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \tag{A.26}$$

By subtracting $(A.26)$ from $(A.25)$, we have

$$f(x+h) - f(x-h) = 2hf'(x) + O(h^3),$$

and consequently

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2). \tag{A.27}$$

Note that the error is now $O(h^2)$ whereas the error term of $(A.23)$ is $O(h)$. In order to see if the error is actually reduced, let us compare the following two approximations

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \text{ and } f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

by applying them to the discrete version of $\sin(x)$ on the interval $(0, \pi)$. As usual, we let $n \geqslant 1$ be a given integer, and define the mesh

$$x_i = ih \text{ for } i = 0, 1, \ldots, n,$$

where $h = \pi/n$. At the nodes, we have the functional values

$$s_i = \sin(x_i) \text{ for } i = 0, 1, \ldots, n,$$

and at the inner nodes we define the first (F) and second (S) order approximations of the derivatives given by

$$d_i^F = \frac{s_{i+1} - s_i}{h},$$

and

$$d_i^S = \frac{s_{i+1} - s_{i-1}}{2h},$$

respectively for $i = 1, 2, \ldots, n-1$. These values should be compared to the exact derivative given by

$$d_i = \cos(x_i) \text{ for } i = 1, 2, \ldots, n-1.$$

The following program, found in `diff_1st2nd_order.py`, plots the discrete functions $(x_i, d_i)_{i=1}^{n-1}$, $(x_i, d_i^F)_{i=1}^{n-1}$, and $(x_i, d_i^S)_{i=1}^{n-1}$ for a given $n$. Note that the first three functions in this program are completely general in that they can be used for any $f(x)$ on any mesh. The special case of $f(x) = \sin(x)$ and comparing first- and second-order formulas is implemented in the `example` function. This latter function is called in the test block of the file. That is, the file is a module and we can reuse the first three functions in other programs (in particular, we can use the third function in the next example).

```
def first_order(f, x, h):
    return (f(x+h) - f(x))/h

def second_order(f, x, h):
    return (f(x+h) - f(x-h))/(2*h)

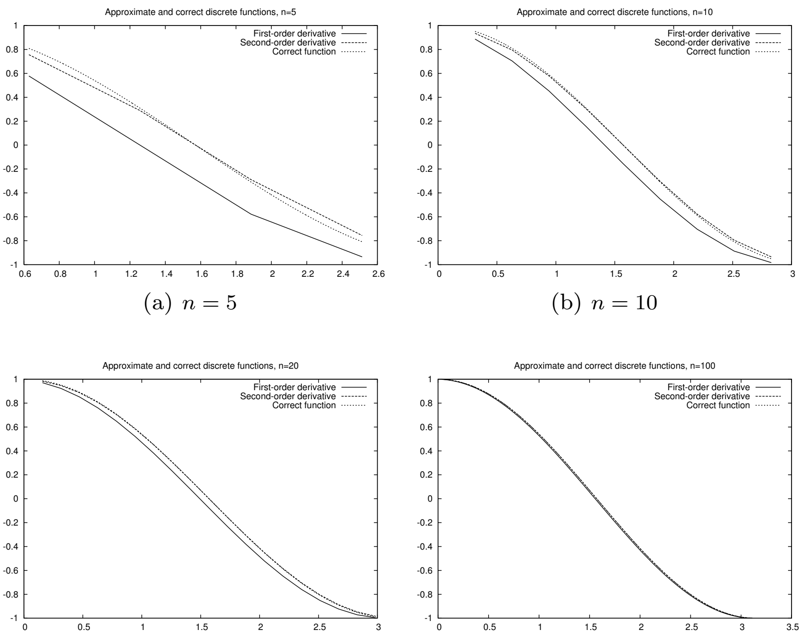
def derivative_on_mesh(formula, f, a, b, n):
    """
    Differentiate f(x) at all internal points in a mesh
    on [a,b] with n+1 equally spaced points.
    The differentiation formula is given by formula(f, x, h).
    """
    h = (b-a)/float(n)
    x = linspace(a, b, n+1)
    df = zeros(len(x))
    for i in xrange(1, len(x)-1):
        df[i] = formula(f, x[i], h)
    # return x and values at internal points only
    return x[1:-1], df[1:-1]

def example(n):
    a = 0; b = pi;
    x, dF = derivative_on_mesh(first_order,  sin, a, b, n)
    x, dS = derivative_on_mesh(second_order, sin, a, b, n)
    # accurate plot of the exact derivative at internal points:
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001)
    exact = cos(xfine)
    plot(x, dF, 'r-', x, dS, 'b-', xfine, exact, 'y-',
         legend=('First-order derivative',
                 'Second-order derivative',
                 'Correct function'),
         title='Approximate and correct discrete '\
               'functions, n=%d' % n)

# main program:
from scitools.std import *
try:
    n = int(sys.argv[1])
```

```
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

example(n)
```




(a) $n = 5$ (b) $n = 10$ (c) $n = 20$ (d) $n = 100$

**Fig. A.4** Plots of exact and approximate derivatives with various number of mesh points $n$.

The result of running the program with four different $n$ values is presented in Figure A.4. Observe that $d_i^S$ is a better approximation to $d_i$ than $d_i^F$, and note that both approximations become very good as $n$ is getting large.

### A.4.7 Second-Order Derivatives

We have seen that the Taylor series can be used to derive approximations of the derivative. But what about higher order derivatives? Next we shall look at second order derivatives. From $(A.20)$ we have

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + O(h^4),$$

and by using $-h$, we have

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2} f''(x_0) - \frac{h^3}{6} f'''(x_0) + O(h^4)$$

By adding these equations, we have

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + O(h^4),$$

and thus

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + O(h^2). \tag{A.28}$$

For a discrete function $(x_i, y_i)_{i=0}^n$, $y_i = f(x_i)$, we can define the following approximation of the second derivative,

$$d_i = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}\,. \tag{A.29}$$

We can make a function, found in the file `diff2nd.py`, that evaluates (A.29) on a mesh. As an example, we apply the function to

$$f(x) = \sin(e^x),$$

where the exact second-order derivative is given by

$$f''(x) = e^x \cos(e^x) - (\sin(e^x))\, e^{2x}\,.$$

```
from diff_1st2nd_order import derivative_on_mesh
from scitools.std import *

def diff2nd(f, x, h):
    return (f(x+h) - 2*f(x) + f(x-h))/(h**2)

def example(n):
    a = 0;  b = pi

    def f(x):
        return sin(exp(x))

    def exact_d2f(x):
        e_x = exp(x)
        return e_x*cos(e_x) - sin(e_x)*exp(2*x)

    x, d2f = derivative_on_mesh(diff2nd, f, a, b, n)
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001)  # fine mesh for comparison
    exact = exact_d2f(xfine)
    plot(x, d2f, 'r-', xfine, exact, 'b-',
         legend=('Approximate derivative',
                 'Correct function'),
         title='Approximate and correct second order '\
               'derivatives, n=%d' % n,
         hardcopy='tmp.eps')

n = int(sys.argv[1])

example(n)
```

In Figure A.5 we compare the exact and the approximate derivatives for $n = 10, 20, 50$, and $100$. As usual, the error descreases when $n$ becomes larger, but note here that the error is very large for small values of $n$.

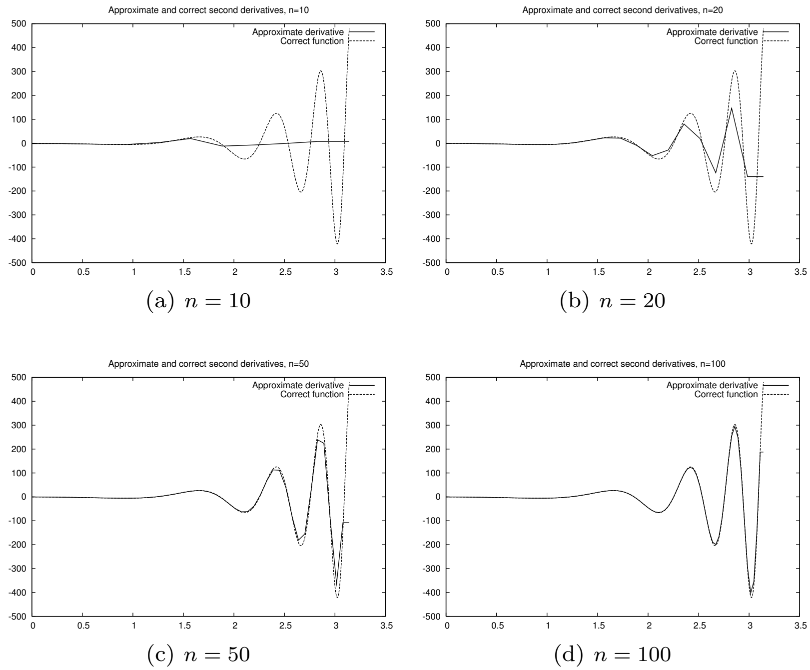



**Fig. A.5** Plots of exact and approximate second-order derivatives with various mesh resolution $n$.

## A.5 Exercises

**Exercise A.1.** *Interpolate a discrete function.*
In a Python function, represent the mathematical function

$$f(x) = \exp\left(-x^2\right)\cos(2\pi x)$$

on a mesh consisting of $q+1$ equally spaced points on $[-1,1]$, and return 1) the interpolated function value at $x = -0.45$ and 2) the error in the interpolated value. Call the function and write out the error for $q = 2, 4, 8, 16$. Name of program file: `interpolate_exp_cos.py` ◇

**Exercise A.2.** *Study a function for different parameter values.*
Develop a program that creates a plot of the function $f(x) = \sin(\frac{1}{x+\varepsilon})$ for $x$ in the unit interval, where $\varepsilon > 0$ is a given input parameter. Use $n+1$ nodes in the plot.

(a) Test the program using $n = 10$ and $\varepsilon = 1/5$.
(b) Refine the program such that it plots the function for two values of $n$; say $n$ and $n + 10$.
(c) How large do you have to choose $n$ in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.

(d) Let $\varepsilon = 1/10$, and repeat (c).
(e) Let $\varepsilon = 1/20$, and repeat (c).
(f) Try to find a formula for how large $n$ needs to be for a given value of $\varepsilon$ such that increasing $n$ further does not change the plot so much that it is visible on the screen. Note that there is no exact answer to this question.

Name of program file: `plot_sin_eps.py` ◇

**Exercise A.3.** *Study a function and its derivative.*
Consider the function

$$f(x) = \sin\left(\frac{1}{x+\varepsilon}\right)$$

for $x$ ranging from 0 to 1, and the derivative

$$f'(x) = \frac{-\cos\left(\frac{1}{x+\varepsilon}\right)}{(x+\varepsilon)^2}.$$

Here, $\varepsilon$ is a given input parameter.

(a) Develop a program that creates a plot of the derivative of $f = f(x)$ based on a finite difference approximation using $n$ computational nodes. The program should also graph the exact derivative given by $f' = f'(x)$ above.
(b) Test the program using $n = 10$ and $\varepsilon = 1/5$.
(c) How large do you have to choose $n$ in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.
(d) Let $\varepsilon = 1/10$, and repeat (c).
(e) Let $\varepsilon = 1/20$, and repeat (c).
(f) Try determine experimentally how large $n$ needs to be for a given value of $\varepsilon$ such that increasing $n$ further does not change the plot so much that you can view it on the screen. Note, again, that there is no exact solution to this problem.

Name of program file: `sin_deriv.py` ◇

**Exercise A.4.** *Use the Trapezoidal method.*
The purpose of this excercise is to test the program `trapezoidal.py`.

(a) Let

$$\bar{a} = \int_0^1 e^{4x}dx = \frac{1}{4}e^4 - \frac{1}{4}.$$

Compute the integral using the program `trapezoidal.py` and, for a given $n$, let $a(n)$ denote the result. Try to find, experimentally, how large you have to choose $n$ in order for

$$|\bar{a} - a(n)| \leqslant \varepsilon$$

where $\varepsilon = 1/100$.

(b) Repeat (a) with $\varepsilon = 1/1000$.

(c) Repeat (a) with $\varepsilon = 1/10000$.

(d) Try to figure out, in general, how large $n$ has to be in order for

$$|\bar{a} - a(n)| \leqslant \varepsilon$$

for a given value of $\varepsilon$.

Name of program file: `trapezoidal_test_exp.py` ◇

**Exercise A.5.** *Compute a sequence of integrals.*

(a) Let

$$\bar{b}_k = \int_0^1 x^k dx = \frac{1}{k+1},$$

and let $b_k(n)$ denote the result of using the program `trapezoidal.py` to compute $\int_0^1 x^k dx$. For $k = 4, 6$ and $8$, try to figure out, by doing numerical experiments, how large $n$ needs to be in order for $b_k(n)$ to satisfy

$$\left|\bar{b}_k - b_k(n)\right| \leqslant 0.0001.$$

Note that $n$ will depend on $k$. Hint: Run the program for each $k$, look at the output, and calculate $\left|\bar{b}_k - b_k(n)\right|$ manually.

(b) Try to generalize the result in (a) to arbitrary $k \geqslant 2$.

(c) Generate a plot of $x^k$ on the unit interval for $k = 2, 4, 6, 8$, and $10$, and try to figure out if the results obtained in (a) and (b) are reasonable taking into account that the program `trapezoidal.py` was developed using a piecewise linear approximation of the function.

Name of program file: `trapezoidal_test_power.py` ◇

**Exercise A.6.** *Use the Trapezoidal method.*

The purpose of this excercise is to compute an approximation of the integral[9]

$$I = \int_{-\infty}^{\infty} e^{-x^2} dx$$

using the Trapezoidal method.

(a) Plot the function $e^{-x^2}$ for $x$ ranging from $-10$ to $10$ and use the plot to argue that

$$\int_{-\infty}^{\infty} e^{-x^2} dx = 2\int_0^{\infty} e^{-x^2} dx.$$

(b) Let $T(n, L)$ be the approximation of the integral

[9] You may consult your Calculus book to verify that the exact solution is $\sqrt{\pi}$.

$$2\int_0^L e^{-x^2}dx$$

computed by the Trapezoidal method using $n$ computational points. Develop a program that computes the value of $T$ for a given $n$ and $L$.

(c) Extend the program developed in (b) to write out values of $T(n, L)$ in a table with rows corresponding to $n = 100, 200, \ldots, 500$ and columns corresponding to $L = 2, 4, 6, 8, 10$.

(d) Extend the program to also print a table of the errors in $T(n, L)$ for the same $n$ and $L$ values as in (c). The exact value of the integral is $\sqrt{\pi}$.

*Comment.* Numerical integration of integrals with finite limits requires a choice of $n$, while with infinite limits we also need to truncate the domain, i.e., choose $L$ in the present example. The accuracy depends on both $n$ and $L$. Name of program file: `integrate_exp.py` ◇

**Exercise A.7.** *Trigonometric integrals.*

The purpose of this excercise is to demonstrate a property of trigonometric functions that you will meet in later courses. In this excercise, you may compute the integrals using the program `trapezoidal.py` with $n = 100$.

(a) Consider the integrals

$$I_{p,q} = 2\int_0^1 \sin(p\pi x)\sin(q\pi x)dx$$

and fill in values of the integral $I_{p,q}$ in a table with rows corresponding to $q = 0, 1, \ldots, 4$ and columns corresponding to $p = 0, 1, \ldots, 4$.

(b) Repeat (a) for the integrals

$$I_{p,q} = 2\int_0^1 \cos(p\pi x)\cos(q\pi x)dx.$$

(c) Repeat (a) for the integrals

$$I_{p,q} = 2\int_0^1 \cos(p\pi x)\sin(q\pi x)dx.$$

Name of program file: `ortho_trig_funcs.py` ◇

**Exercise A.8.** *Plot functions and their derivatives.*

(a) Use the program `diff_func.py` to plot approximations of the derivative for the following functions defined on the interval ranging from $x = 1/1000$ to $x = 1$:

$$f(x) = \ln\left(x + \frac{1}{100}\right),$$
$$g(x) = \cos(e^{10x}),$$
$$h(x) = x^x.$$

(b) Extend the program such that both the discrete approximation and the correct (analytical) derivative can be plotted. The analytical derivative should be evaluated in the same computational points as the numerical approximation. Test the program by comparing the discrete and analytical derivative of $x^3$.

(c) Use the program develped in (b) to compare the analytical and discrete derivatives of the functions given in (a). How large do you have to choose $n$ in each case in order for the plots to become indistinguishable on your screen. Note that the analytical derivatives are given by:

$$f'(x) = \frac{1}{x + \frac{1}{100}},$$
$$g'(x) = -10e^{10x}\sin\left(e^{10x}\right)$$
$$h'(x) = (\ln x)\,x^x + x x^{x-1}$$

Name of program file: `diff_functions.py` ◇

**Exercise A.9.** *Use the Trapezoidal method.*
Develop an efficient program that creates a plot of the function

$$f(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

for $x \in [0, 10]$. The integral should be approximated using the Trapezoidal method and use as few function evaluations of $e^{-t^2}$ as possible.
Name of program file: `plot_integral.py` ◇

# B Differential Equations

**This appendix is authored by Aslak Tveito**

Differential equations have proven to be an immensely successful instrument for modeling phenomena in science and technology. It is hardly an exaggeration to say that differential equations are used to define mathematical models in virtually all parts of the natural sciences. In this chapter, we will take the first steps towards learning how to deal with differential equations on a computer. This is a core issue in Computational Science and reaches far beyond what we can cover in this text. However, the ideas you will see here are reused in lots of advanced applications, so this chapter will hopefully provide useful introduction to a topic that you will probably encounter many times later.

We will show you how to build programs for solving differential equations. More precisely, we will show how a differential equation can be formulated in a discrete manner suitable for analysis on a computer, and how to implement programs to compute the discrete solutions. The simplest differential equations can be solved analytically in the sense that you can write down an explicit formula for the solutions. However, differential equations arising in practical applications are usually rather complicated and thus have to be solved numerically on a computer. Therefore we focus on implementing numerical methods to solve the equations. Chapters 7.4 and 9.4 describe more advanced implementation techniques aimed at making an easy-to-use toolbox for solving differential equations. Exercises in this appendix and the mentioned chapters aim at solving a variety of differential equations arising in various disciplines of science.

As with all the other chapters, the source code can be found in `src`, in this case in the subdirectory `ode` (the short form ODE is commonly

used as abbreviation for “Ordinary Differential Equations”, which is the type of differential equation that we primarily address in this chapter).

## B.1 The Simplest Case

Consider the problem of solving the following equation

$$u'(t) = t^3\,. \tag{B.1}$$

The solution can be computed directly by integrating (B.1), which gives

$$u(t) = \frac{1}{4}t^4 + C,$$

where $C$ is an arbitrary constant. To obtain a unique solution, we need an extra condition to determine $C$. Specifying $u(t_1)$ for some time point $t_1$ represents a possible extra condition. It is common to view (B.1) as an equation for the function $u(t)$ for $t \in [0, T]$, and the extra condition is usually that the start value $u(0)$ is known. This is called the *initial condition*. Say

$$u(0) = 1\,. \tag{B.2}$$

In general, the solution of the differential equation (B.1) subject to the initial condition B.2 is[1]

$$\begin{aligned} u(t) &= u(0) + \int_0^t u'(\tau)d\tau, \\ &= 1 + \int_0^t \tau^3 d\tau \\ &= 1 + \frac{1}{4}t^4\,. \end{aligned}$$

Let us go back and check: Does $u(t) = 1 + \frac{1}{4}t^4$ really satisfy the two requirements listed in $(B.1)$ and $(B.2)$? Obviously, $u(0) = 1$, and $u'(t) = t^3$, so the solution is correct.

More generally, we consider the equation

$$u'(t) = f(t) \tag{B.3}$$

together with the initial condition

$$u(0) = u_0. \tag{B.4}$$

Here we assume that $f(t)$ is a given function, and that $u_0$ is a given number. Then, by reasoning as above, we have

[1] If you are confused by the use of $t$ and $\tau$, don’t get too upset; you see: ”In mathematics you don’t understand things. You just get used to them.” –John von Neumann, mathematician, 1903-1957.

$$u(t) = u_0 + \int_0^T f(\tau)d\tau. \tag{B.5}$$

By using the methods introduced in Appendix A, we can find a discrete version of $u$ by approximating the integral. Generally, an approximation of the integral

$$\int_0^T f(\tau)d\tau$$

can be computed using the discrete version of a continuous function $f(\tau)$ defined on an interval $[0, t]$ . The discrete version of $f$ is given by $(\tau_i, y_i)_{i=0}^n$ where

$$\tau_i = ih, \text{ and } y_i = f(\tau_i)$$

for $i = 0, 1, \ldots, n$. Here $n \geqslant 1$ is a given integer and $h = T/n$. The Trapezoidal rule can now be written as

$$\int_0^T f(\tau)d\tau \approx \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right]. \tag{B.6}$$

By using this approximation, we find that an approximate solution of (B.3)–(B.4) is given by

$$u(t) \approx u_0 + \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right].$$

The program `integrate_ode.py` computes a numerical solution of (B.3)–(B.4), where the function $f$, the time $t$, the initial condtion $u_0$, and the number of time-steps $n$ are inputs to the program.

```
def integrate(T, n, u0):
    h = T/float(n)
    t = linspace(0, T, n+1)
    I = f(t[0])
    for k in iseq(1, n-1, 1):
        I += 2*f(t[k])
    I += f(t[-1])
    I *= (h/2)
    I += u0
    return I

from scitools.std import *
f_formula = sys.argv[1]
T  = eval(sys.argv[2])
u0 = eval(sys.argv[3])
n  = int(sys.argv[4])

f = StringFunction(f_formula, independent_variables='t')
print "Numerical solution of u'(t)=t**3: ", integrate(T, n, u0)
```

We apply the program for computing the solution of

$$u'(t) = te^{t^2},$$
$$u(0) = 0,$$

at time $T = 2$ using $n = 10, 20, 50$ and $100$:

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 10
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3:  28.4066160877
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 20
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3:  27.2059977451
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 50
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3:  26.86441489
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 100
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3:  26.8154183399
```

The exact solution is given by $\frac{1}{2}e^{2^2} - \frac{1}{2} \approx 26.799$, so we see that the approximate solution becomes better as $n$ is increased, as expected.

## B.2 Exponential Growth

The example above was really not much of a differential equation, because the solution was obtained by straightforward integration. Equations of the form

$$u'(t) = f(t) \tag{B.7}$$

arise in situations where we can explicitly specify the derivative of the unknown function $u$. Usually, the derivative is specified in terms of the solution itself. Consider, for instance, population growth under idealized conditions as modeled in Chapter 5.1.4. We introduce the symbol $v_i$ for the number of individuals at time $\tau_i$ ($v_i$ corresponds to $x_n$ in Chapter 5.1.4). The basic model for the evolution of $v_i$ is (5.9):

$$v_i = (1 + r)v_{i-1}, \quad i = 1, 2, \ldots, \text{ and } v_0 \text{ known}. \tag{B.8}$$

As mentioned in Chapter 5.1.4, $r$ depends on the time difference $\Delta\tau = \tau_i - \tau_{i-1}$: the larger $\Delta\tau$ is, the larger $r$ is. It is therefore natural to

introduce a growth rate $\alpha$ that is independent of $\Delta\tau$: $\alpha = r/\Delta\tau$. The number $\alpha$ is then fixed regardless of how long jumps in time we take in the difference equation for $v_i$. In fact, $\alpha$ equals the growth in percent, divided by 100, over a time interval of unit length.

The difference equation now reads

$$v_i = v_{i-1} + \alpha\Delta\tau\, v_{i-1}\,.$$

Rearring this equation we get

$$\frac{v_i - v_{i-1}}{\Delta\tau} = \alpha v_{i-1}\,. \tag{B.9}$$

Assume now that we shrink the time step $\Delta\tau$ to a small value. The left-hand side of (B.9) is then an approximation to the time-derivative of a function $v(\tau)$ expressing the number of individuals in the population at time $\tau$. In the limit $\Delta\tau \to 0$, the left-hand side becomes the derivative exactly, and the equation reads

$$v'(\tau) = \alpha v(\tau)\,. \tag{B.10}$$

As for the underlying difference equation, we need a start value $v(0) = v_0$. We have seen that reducing the time step in a difference equation to zero, we get a differential equation.

Many like to scale an equation like (B.10) such that all variables are without physical dimensions and their maximum absolute value is typically of the order of unity. In the present model, this means that we introduce new dimensionless variables

$$u = \frac{v}{v_0}, \quad t = \frac{\tau}{\alpha}$$

and derive an equation for $u(t)$. Inserting $v = v_0 u$ and $\tau = \alpha t$ in (B.10) gives the prototype equation for population growth:

$$u'(t) = u(t) \tag{B.11}$$

with the intial condition

$$u(0) = 1\,. \tag{B.12}$$

When we have computed the dimensionless $u(t$, we can find the function $v(\tau)$ as

$$v(\tau) = v_0 u(\tau/\alpha)\,.$$

We shall consider practical applications of population growth equations later, but let's start by looking at the idealized case (B.11).

*Analytical Solution.* Our differential equation can be written in the form

$$\frac{du}{dt} = u,$$

which can be rewritten as

$$\frac{du}{u} = dt,$$

and then integration on both sides yields

$$\ln(u) = t + c,$$

where $c$ is a constant that has to be determined by using the initial condition. Putting $t = 0$, we have

$$\ln(u(0)) = c,$$

hence

$$c = \ln(1) = 0,$$

and then

$$\ln(u) = t,$$

so we have the solution

$$u(t) = e^t. \tag{B.13}$$

Let us now check that this function really solves $(B.7, B.11)$. Obviously, $u(0) = e^0 = 1$, so $(B.11)$ is fine. Furthermore

$$u'(t) = e^t = u(t),$$

thus $(B.7)$ also holds.

*Numerical Solution.* We have seen that we can find a formula for the solution of the equation of exponential growth. So the problem is solved, and it is trivial to write a program to graph the solution. We will, however, go one step further and develop a numerical solution strategy for this problem. We don't really need such a method for this problem since the solution is available in terms of a formula, but as mentioned earlier, it is good practice to develop methods for problems where we know the solution; then we are more confident when we are confronted with more challenging problems.

Suppose we want to compute a numerical approximation of the solution of

$$u'(t) = u(t) \tag{B.14}$$

equipped with the intial condition

$$u(0) = 1. \tag{B.15}$$

We want to compute approximations from time $t = 0$ to time $t = 1$. Let $n \geqslant 1$ be a given integer, and define

$$\Delta t = 1/n\,. \tag{B.16}$$

Furthermore, let $u_k$ denote an approximation of $u(t_k)$ where

$$t_k = k\Delta t \tag{B.17}$$

for $k = 0, 1, \ldots, n$. The key step in developing a numerical method for this differential equation is to invoke the Taylor series as applied to the exact solution,

$$u(t_{k+1}) = u(t_k) + \Delta t u'(t_k) + O(\Delta t^2), \tag{B.18}$$

which implies that

$$u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}\,. \tag{B.19}$$

By using $(B.14)$ , we get

$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} \approx u(t_k). \tag{B.20}$$

Recall now that $u(t_k)$ is the exact solution at time $t_k$, and that $u_k$ is the approximate solution at the same point in time. We now want to determine $u_k$ for all $k \geqslant 0$. Obviously, we start by defining

$$u_0 = u(0) = 1.$$

Since we want $u_k \approx u(t_k)$, we require that $u_k$ satisfy the following equality

$$\frac{u_{k+1} - u_k}{\Delta t} = u_k \tag{B.21}$$

motivated by $(B.20)$ . It follows that

$$u_{k+1} = (1 + \Delta t)u_k\,. \tag{B.22}$$

Since $u_0$ is known, we can compute $u_1, u_2$ and so on by using the formula above. The formula is implemented[2] in the program `exp_growth.py`.

```
def compute_u(u0, T, n):
    """Solve u'(t)=u(t), u(0)=u0 for t in [0,T] with n steps."""
    u = u0
    dt = T/float(n)
    for k in range(0, n, 1):
        u = (1+dt)*u
    return u  # u(T)
```

[2] Actually, we do not need the method and we do not need the program. It follows from $(B.22)$ that

$$u_k = (1 + \Delta t)^k u_0$$

for $k = 0, 1, \ldots, n$ which can be evaluated on a pocket calculator or even on your cellular phone. But again, we show examples where everything is as simple as possible (but not simpler!) in order to prepare your mind for more complex matters ahead.

```
import sys
n  = int(sys.argv[1])

# special test case: u'(t)=u, u(0)=1, t in [0,1]
T = 1; u0 = 1
print 'u(1) =', compute_u(u0, T, n)
```

Observe that we do not store the $u$ values: We just overwrite a `float` object `u` by its new value. This saves a lot of storage if $n$ is large.

Running the program for $n = 5, 10, 20$ and 100, we get the approximations 2.4883, 2.5937, 2.6533, and 2.7048. The exact solution at time $t = 1$ is given by $u(1) = e^1 \approx 2.7183$, so again the approximations become better as $n$ is increased.

An alternative program, where we plot $u(t)$ and therefore store all the $u_k$ and $t_k = k\Delta t$ values, is shown below.

```
def compute_u(u0, T, n):
    """Solve u'(t)=u(t), u(0)=u0 for t in [0,T] with n steps."""
    t = linspace(0, T, n+1)
    t[0] = 0
    u = zeros(n+1)
    u[0] = u0
    dt = T/float(n)
    for k in range(0, n, 1):
        u[k+1] = (1+dt)*u[k]
        t[k+1] = t[k] + dt
    return u, t

from scitools.std import *
n  = int(sys.argv[1])

# special test case: u'(t)=u, u(0)=1, t in [0,1]
T = 1; u0 = 1
u, t = compute_u(u0, T, n)
plot(t, u)
tfine = linspace(0, T, 1001) # for accurate plot
v = exp(tfine)               # correct solution
hold('on')
plot(tfine, v)
legend(['Approximate solution', 'Correct function'])
title('Approximate and correct discrete functions, n=%d' % n)
hardcopy('tmp.eps')
```

Using the program for $n = 5, 10, 20$, and 100, results in the plots in Figure B.1. The convergence towards the exponential function is evident from these plots.

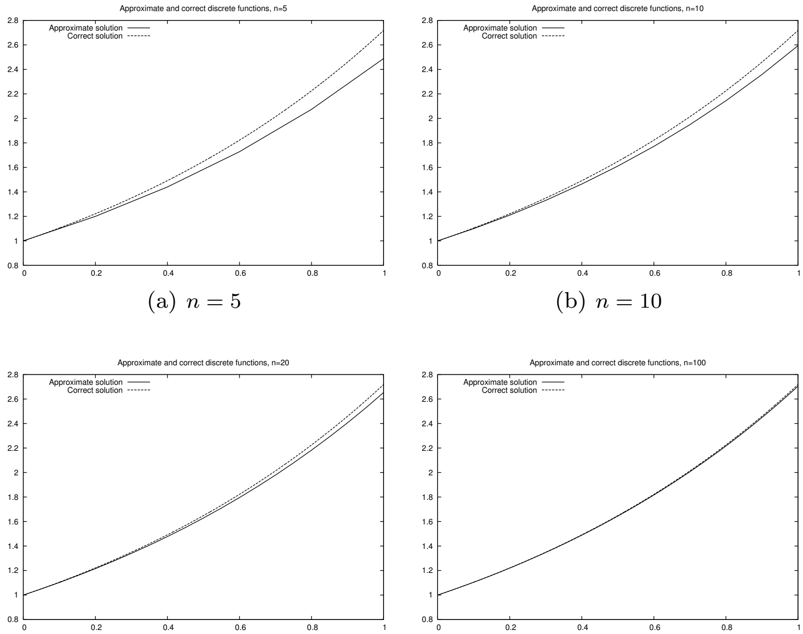
## B.3 Logistic Growth

Exponential growth can be modelled by the following equation

$$u'(t) = \alpha u(t)$$

where $a > 0$ is a given constant. If the initial condition is given by

$$u(0) = u_0$$




(a) $n = 5$ (b) $n = 10$

(c) $n = 20$ (d) $n = 100$

**Fig. B.1** Plots of exact and approximate solutions of $u'(t) = u(t)$ with varying number of time steps in $[0, 1]$..

the solution is given by

$$u(t) = u_0 e^{\alpha t}.$$

Since $a > 0$, the solution becomes very large as $t$ increases. For a short time, such growth of a population may be realistic, but over a longer time, the growth of a population is restricted due to limitations of the environment, as discussed in Chapter 5.1.5. Introducing a logistic growth term as in (5.12) we get the differential equation

$$u'(t) = \alpha u(t) \left(1 - \frac{u(t)}{R}\right), \tag{B.23}$$

where $\alpha$ is the growth-rate, and $R$ is the carrying capacity (which corresponds to $M$ in Chapter 5.1.5). Note that $R$ is typically very large, so if $u(0)$ is small, we have

$$\frac{u(t)}{R} \approx 0$$

for small values of $t$, and thus we have exponential growth for small $t$;

$$u'(t) \approx au(t).$$

But as $t$ increases, and $u$ grows, the term $u(t)/R$ will become important and limit the growth.

A numerical scheme for the logistic equation ($B.23$) is given by

$$\frac{u_{k+1} - u_k}{\Delta t} = \alpha u_k \left(1 - \frac{u_k}{R}\right),$$

which we can solve with respect to the unknown $u_{k+1}$:

$$u_{k+1} = u_k + \Delta t \alpha u_k \left(1 - \frac{u_k}{R}\right). \tag{B.24}$$

This is the form of the equation that is suited for implementation.

## B.4 A General Ordinary Differential Equation

Let us briefly consider a general ordinary[3] differential equations on the form

$$u'(t) = f(u(t)) \tag{B.25}$$

subject to the initial condition

$$u(0) = u_0$$

where $f(u)$ is a given function and the initial state $u_0$ is given. Suppose we want to compute an approximate solution of (B.25) for $t$ ranging from $t = 0$ to $t = T$ where $T > 0$ is given. As above we start by introducing the time step

$$\Delta t = T/n.$$

where $n \geqslant 1$ is a given integer, and we let $u_k$ denote an approximation of the exact solution $u(t_k)$. Also as above, we replace the derivative of $u$ with a finite difference and obtain the scheme

$$\frac{u_{k+1} - u_k}{\Delta t} = f(u_k).$$

The scheme can be rewritten in a form that is more suitable for computations;

[3] Differential equations are divided into two groups: ordinary differential equations and partial differential equations. Ordinary differential equations contain derivatives with respect to one variable (usually $t$ in our examples), whereas partial differential equations contain derivatives with respect to more than one variable, typically with respect to space and time. A typical ordinary differential equation is

$$u'(t) = u(t),$$

and a typical partial differential equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

The latter is known as the heat or diffusion equation.

$$u_{k+1} = u_k + \Delta t f(u_k). \tag{B.26}$$

We note that since $u_0$ is known, we can compute $u_1, u_2$ and so on. This scheme is commonly referred to as the Explicit Euler[4] scheme, or the Forward Euler scheme, or the Forward Euler method. We will later derive the Implicit Euler scheme, also called the Backward Euler scheme (or method). That scheme is sligtly harder to use but it has some nice properties that will be discussed later.

The Explicit Euler scheme is implemented in the function `Explicit_Euler` in the module `Euler`. The test block in the module file allows input from the command line: the formula for $f(u)$, the number of time steps $n$, the final time $T$, and the initial condition $u_0$.

```
from numpy import linspace, zeros

def Explicit_Euler(f, u0, T, n):
    dt = T/float(n)
    t = linspace(0, T, n+1)
    u = zeros(n+1)
    u[0] = u0
    for k in range(n):
        u[k+1] = u[k] + dt*f(u[k])
    return u, t

if __name__ == '__main__':
    f_formula = sys.argv[1]
    n = int(sys.argv[2])
    T = eval(sys.argv[3])
    u0 = eval(sys.argv[4])

    f = StringFunction(f_formula, independent_variables='u')
    u, t = Explicit_Euler(f, u0, T, n)
    plot(t, u)
```

In Figure B.2 we see the results of the program as applied to the problem

$$u' = e^u$$

with $u_0 = 0$, $T = 1$. The numerical results are provided for $n = 5, 10, 20$ and 100. Convergence is not as obvious anymore, so let us also try the program for $n = 100, 200, 300$ and 400. The results are given in Figure B.3 and we see that the approximations seem to tend to a common limiting function.

## B.5 A Simple Pendulum

So far we have considered scalar ordinary differential equations, i.e., equations with one single function $u(t)$ as unknown. Now we shall deal

[4] Leonhard Paul Euler, 1707–1783. A pioneering Swiss mathematician and physicist who spent most of his life in Russia and Germany. Euler is one of the greatest scientists of all time, and made important contributions to calculus, mechanics, optics, and astronomy. He also introduced much of the modern terminology and notation in mathematics.

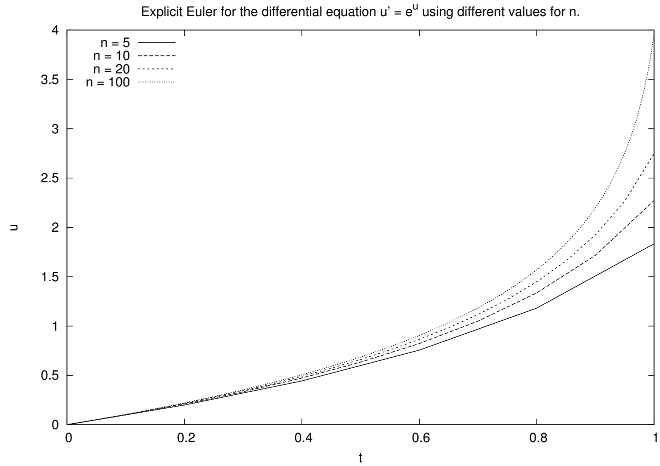



**Fig. B.2** Explicit Euler for the differential equation $u' = e^u$ using different values for n.

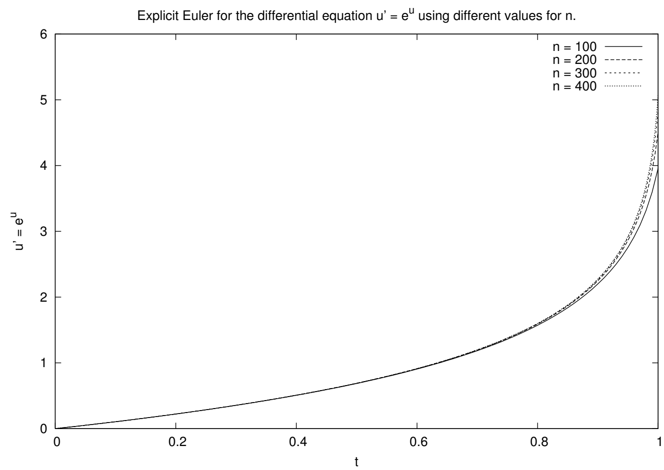



**Fig. B.3** Explicit Euler for the differential equation $u' = e^u$ using different values for $n$.

with systems of ordinary differential equations, where in general $n$ unknown functions are coupled in a system of $n$ equations. Our introductory example will be a system of two equations having two unknown functions $u(t)$ and $v(t)$. The example concerns the motion of a pendulum, see Figure B.4. A sphere with mass $m$ is attached to a massless rod of length $L$ and oscillates back and forth due to gravity. Newton's second law of motion applied to this physical system gives rise the differential equation

$$\theta''(t) + \alpha \sin(\theta) = 0 \tag{B.27}$$

where $\theta = \theta(t)$ is the angle the rod makes with the vertical, measured in radians, and $\alpha = g/L$ ($g$ is the acceleration of gravity). The un-

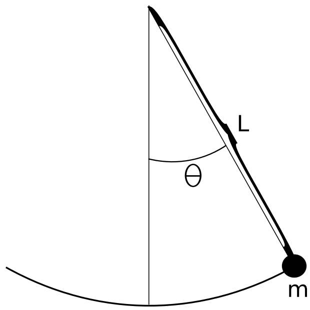



**Fig. B.4** A pendulum with $m$ = mass, $L$ = length of massless rod and $\theta = \theta(t)$ = angle.

known function to solve for is $\theta$, and knowing $\theta$, we can quite easily compute the position of the sphere, its velocity, and its acceleration, as well as the tension force in the rod. Since the highest derivative in (B.27) is of second order, we refer to (B.27) as a *second-order differential equations*. Our previous examples in this chapter involved only first-order derivatives, and therefore they are known as *first-order differential equations*.

Equation (B.27) can be solved by the same numerical method as we use in Appendix C.1.2, because (B.27) is very similar to Equation C.8, which is the topic of Appendix C. The only difference is that C.8 has extra terms, which can be skipped, while the $kS$ term in C.8 must be extended to $\alpha \sin(S)$ to make C.8 identical to (B.27). This extension is easily performed. However, here we shall not solve the second-order equation (B.27) as it stands. We shall instead rewrite it as a system of two first-order equations so that we can use numerical methods for first-order equations to solve it.

To transform a second-order equation to a system of two first-order equations, we introduce a new variable for the first-order derivative (the angular velocity of the sphere): $v(t) = \theta'(t)$. Using $v$ and $\theta$ in (B.27) yields

$$v'(t) + \alpha \sin(\theta) = 0\,.$$

In addition, we have the relation

$$v = \theta'(t)$$

between $v$ and $\theta$. This means that (B.27) is equivalent to the following system of two coupled first-order differential equations:

$$\theta'(t) = v(t), \tag{B.28}$$

$$v'(t) = -\alpha \sin(\theta)\,. \tag{B.29}$$

As for scalar differential equations, we need initial conditions, now two conditions because we have two unknown functions:

$$\theta(0) = \theta_0,$$
$$v(0) = v_0,$$

Here we assume the initial angle $\theta_0$ and the initial angular velocity $v_0$ to be given.

It is common to group the unknowns and the initial conditions in 2-vectors: $(\theta(t), v(t))$ and $(\theta_0, v_0)$. One may then view (B.28)–(B.29) as a *vector equation*, whose first component equation is (B.28), and the second component equation is (B.29). In Python software, this vector notation makes solution methods for scalar equations (almost) immediately available for vector equations, i.e., systems of ordinary differential equations.

In order to derive a numerical method for the system (B.28)–(B.29), we proceed as we did above for one equation with one unknown function. Say we want to compute the solution from $t = 0$ to $t = T$ where $T > 0$ is given. Let $n \geqslant 1$ be a given integer and define the time step

$$\Delta t = T/n.$$

Furthermore, we let $(\theta_k, v_k)$ denote approximations of the exact solution $(\theta(t_k), v(t_k))$ for $k = 0, 1, \ldots, n$. A Forward Euler type of method will now read

$$\frac{\theta_{k+1} - \theta_k}{\Delta t} = v_k, \tag{B.30}$$
$$\frac{v_{k+1} - v_k}{\Delta t} = -\alpha \sin(\theta_k). \tag{B.31}$$

This scheme can be rewritten in a form more suitable for implementation:

$$\theta_{k+1} = \theta_k + \Delta t\, v_k, \tag{B.32}$$
$$v_{k+1} = v_k - \alpha \Delta t \sin(\theta_k)\,. \tag{B.33}$$

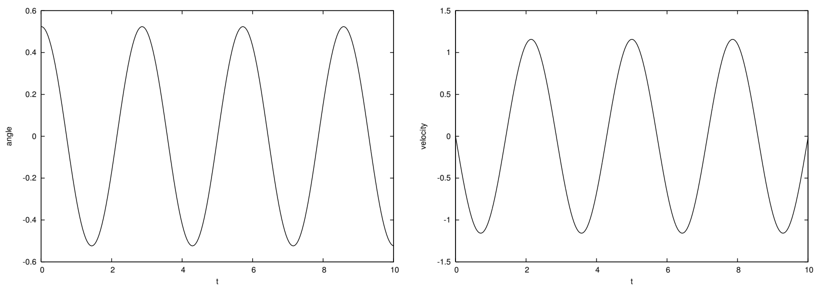
The next program, `pendulum.py`, implements this method in the function `pendulum`. The input parameters to the model, $\theta_0$, $v_0$,, the final time $T$, and the number of time-steps $n$, must be given on the command line.

```
def pendulum(T, n, theta0, v0, alpha):
    """Return the motion (theta, v, t) of a pendulum."""
    dt = T/float(n)
    t = linspace(0, T, n+1)
    v = zeros(n+1)
    theta = zeros(n+1)
    v[0] = v0
    theta[0] = theta0
    for k in range(n):
        theta[k+1] = theta[k] + dt*v[k]
        v[k+1] = v[k] - alpha*dt*sin(theta[k+1])
    return theta, v, t
```

```
from scitools.std import *
n = int(sys.argv[1])
T = eval(sys.argv[2])
v0 = eval(sys.argv[3])
theta0 = eval(sys.argv[4])
alpha = eval(sys.argv[5])

theta, v, t = pendulum(T, n, theta0, v0)
plot(t, v, xlabel='t', ylabel='velocity')
figure()
plot(t, theta, xlabel='t', ylabel='velocity')
```

By running the program with the input data $\theta_0 = \pi/6$, $v_0 = 0$, $\alpha = 5$, $T = 10$ and $n = 1000$, we get the results shown in Figure B.5. The angle $\theta = \theta(t)$ is displayed in the left panel and the velocity is given in the right panel.




(a) (b)

**Fig. B.5** Motion of a pendulum: (a) the angle $\theta(t)$, and (b) the angular velocity $\theta'$.

## B.6 A Model for the Spread of a Disease

Mathematical models are used intensively to analyze the spread of infectious diseases[5]. In the simplest case, we may consider a population, that is supposed to be constant, consisting of two groups; the susceptibles ($S$) who can catch the disease, and the infectives ($I$) who have the disease and are able to transmit it. A system of differential equations modelling the evolution of $S$ and $I$ is given by

$$S' = -rSI,$$
$$I' = rSI - aI.$$

Here $r$ and $a$ are given constants reflecting the characteristics of the epidemic. The initial conditions are given by

[5] The interested reader may consult the excellent book [10] on Mathematical Biology by J.D. Murray for an introduction to such models.

$$S(0) = S_0,$$
$$I(0) = I_0,$$

where the initial state $(S_0, I_0)$ is assumed to be known.

Suppose we want to compute numerical solutions of this system from time $t = 0$ to $t = T$. Then, by reasoning as above, we introduce the time step

$$\Delta t = T/n$$

and the approximations $(S_k, I_k)$ of the solution $(S(t_k), I(t_k))$. An explicit Forward Euler method for the system takes the following form,

$$\frac{S_{k+1} - S_k}{\Delta t} = -rS_k I_k,$$
$$\frac{I_{k+1} - I_k}{\Delta t} = rS_k I_k - aI_k,$$

which can be rewritten on computational form

$$S_{k+1} = S_k - \Delta t r S_k I_k,$$
$$I_{k+1} = I_k + \Delta t \left( rS_k I_k - aI_k \right) .$$

This scheme is implemented in the program `exp_epidemic.py` where $r, a, S_0, I_0, n$ and $T$ are input data given on the command line. The function `epidemic` computes the solution $(S, I)$ to the differential equation system. This pair of time-dependent functions is then plotted in two separate plots.

```
def epidemic(T, n, S0, I0, r, a):
    dt = T/float(n)
    t = linspace(0, T, n+1)
    S = zeros(n+1)
    I = zeros(n+1)
    S[0] = S0
    I[0] = I0
    for k in range(n):
        S[k+1] = S[k] - dt*r*S[k]*I[k]
        I[k+1] = I[k] + dt*(r*S[k]*I[k] - a*I[k])
    return S, I, t

from scitools.std import *
n = int(sys.argv[1])
T = eval(sys.argv[2])
S0 = eval(sys.argv[3])
I0 = eval(sys.argv[4])
r = eval(sys.argv[5])
a = eval(sys.argv[6])

plot(t, S, xlabel='t', ylabel='Susceptibles')
plot(t, I, xlabel='t', ylabel='Infectives')
```

We want to apply the program to a specific case where an influenza epidemic hit a British boarding school with a total of 763 boys[6]. The

[6] The data are from Murray [10], and Murray found the data in the British Medical Journal, March 4, 1978.

epidemic lasted from 21st January to 4th February in 1978. We let $t = 0$ denote 21st of January and we define $T = 14$ days. We put $S_0 = 762$ and $I_0 = 1$ which means that one person was ill at $t = 0$. In the Figure B.6 we se the numerical results using $r = 2.18 \times 10^{-3}, a = 0.44, n = 1000$. Also, we have plotted actual the measurements, and we note that the simulations fit the real data quite well.

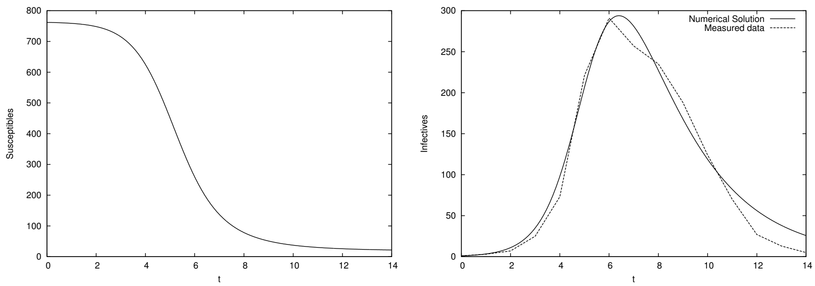



**Fig. B.6** Graphs of (a) susceptibles and (b) infectives for an influenza in a British boarding school in 1978.

## B.7 Exercises

**Exercise B.1.** *Solve a nonhomogeneous linear ODE.*
Solve the ODE problem

$$u' = 2u - 1, \quad u(0) = 2, \quad t \in [0, 6]$$

using the Forward Euler method. Choose $\Delta t = 0.25$. Plot the numerical solution together with the exact solution $u(t) = \frac{1}{2} + 2e^{2t}$. Name of program file: `nonhomogeneous_linear_ODE.py`. ◇

**Exercise B.2.** *Solve a nonlinear ODE.*
Solve the ODE problem

$$u' = u^q, \quad u(0) = 1, \quad t \in [0, T]$$

using the Forward Euler method. The exact solution reads $u(t) = e^t$ for $q = 1$ and $u(t) = (t(1-q)+1)^{1/(1-q)}$ for $q > 1$ and $t(1-q)+1 > 0$. Read $q$, $\Delta t$, and $T$ from the command line, solve the ODE, and plot the numerical and exact solution. Run the program for different cases: $q = 2$ and $q = 3$, with $\Delta t = 0.01$ and $\Delta t = 0.1$. Set $T = 6$ if $q = 1$ and $T = 1/(q-1) - 0.1$ otherwise. Name of program file: `nonlinear_ODE.py`.
◇

**Exercise B.3.** *Solve an ODE for $y(x)$.*

We have given the following ODE problem:

$$\frac{dy}{dx} = \frac{1}{2(y-1)}, \quad y(0) = 1 + \sqrt{\epsilon}, \quad x \in [0, 4], \tag{B.34}$$

where $\epsilon > 0$ is a small number. Formulate a Forward Euler method for this ODE problem and compute the solution for varying step size in $x$: $\Delta x = 1$, $\Delta x = 0.25$, $\Delta x = 0.01$. Plot the numerical solutions together with the exact solution $y(x) = 1 + \sqrt{x + \epsilon}$, using 1001 $x$ coordinates for accurate resolution of the latter. Set $\epsilon$ to $10^{-3}$. Study the numerical solution with $\Delta x = 1$, and use that insight to explain why this problem is hard to solve numerically. Name of program file: `yx_ODE.py`. ◇

**Exercise B.4.** *Experience instability of an ODE.*

Consider the ODE problem

$$u' = \alpha u, \quad u(0) = u_0,$$

solved by the Forward Euler method. Show by repeatedly applying the scheme that

$$u_k = (1 + \alpha \Delta t)^k u_0 \,.$$

We now turn to the case $\alpha < 0$. Show that the numerical solution will oscillate if $\Delta t > -1/\alpha$. Make a program for computing $u_k$, set $\alpha = -1$, and demonstrate oscillatory solutions for $\Delta t = 1.1, 1.5, 1.9$. Recall that that the exact solution, $u(t) = e^{\alpha t}$, never oscillates.

What happens if $\Delta t > -2/\alpha$? Try it out in the program and explain then mathematically why not $u_k \to 0$ as $k \to \infty$. Name of program file: `unstable_ODE.py`. ◇

**Exercise B.5.** *Solve an ODE for the arc length.*

Given a curve $y = f(x)$, the length of the curve from $x = x_0$ to some point $x$ is given by the function $s(x)$, which fulls the problem

$$\frac{ds}{dx} = \sqrt{1 + [f'(x)]^2}, \quad s(x_0) = 0 \,. \tag{B.35}$$

Since $s$ does not enter the right-hand side, (B.35) can immediately be integrated from $x_0$ to $x$. However, we shall solve (B.35) as an ODE. Use the Forward Euler method and compute the length of a straight line (for verification) and a sine curve: $f(x) = \frac{1}{2}x + 1$, $x \in [0, 2]$; $f(x) = \sin(\pi x)$, $x \in [0, 2]$. Name of program file: `arclength_ODE.py`. ◇

**Exercise B.6.** *Solve an ODE with time-varying growth.*

Consider the ODE for exponential growth,

$$u' = \alpha u, \quad u(0) = 1, \quad t \in [0, T] \,.$$

Now we introduce a time-dependent $\alpha$ such that the growth decreases with time: $\alpha(t) = a - bt$. Solve the problem for $a = 1$, $b = 0.1$, and $T = 10$. Plot the solution and compare with the corresponding exponential growth using the mean value of $\alpha(t)$ as growth factor: $e^{(a-bT/2)t}$. Name of program file: `time_dep_growth.py`. ◇

**Exercise B.7.** *Solve an ODE for emptying a tank.*

A cylindrical tank of radius $R$ is filled with water to a height $h(t)$. By opening a valve of radius $r$ at the bottom of the tank, water flows out, and $h(t)$ decreases with time. We can derive an ODE that governs the height function $h(t)$.

Mass conservation of water requires that the reduction in height balances the outflow. In a time interval $\Delta t$, the height is reduced by $\Delta h$, which corresponds to a water volume of $\pi R^2 \Delta h$. The water leaving the tank in the same interval of time equals $\pi r^2 v \Delta t$, where $v$ is the outflow velocity. It can be shown (from what is known as Bernoulli's equation) that

$$v(t) = \sqrt{2gh(t) - h'(t)^2},$$

$g$ being the acceleration of gravity [6, 11]. Letting $\Delta h > 0$ correspond to an increase in $h$, we have that the $-\pi R^2 \Delta h$ must balance $\pi r^2 v \Delta t$, which in the limit $\Delta t \to 0$ leads to the ODE

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \left(1 + \left(\frac{r}{R}\right)^4\right)^{-1/2} \sqrt{2gh}\,. \tag{B.36}$$

A proper initial condition follows from the initial height of water, $h_0$, in the tank: $h(0) = h_0$.

Solve (B.36) in a program using the Forward Euler scheme. Set $r = 1$ cm, $R = 20$ cm, $g = 9.81$ m/s$^2$, and $h_0 = 1$ m. Use a time step of 10 seconds. Plot the solution, and experiment to see what a proper time interval for the simulation is. Can you find an analytical solution of the problem to compare the numerical solution with? Name of program file: `tank_ODE.py`. ◇

**Exercise B.8.** *Solve an ODE system for an electric circuit.*

An electric circuit with a resistor, a capacitor, an inductor, and a voltage source can be described by the ODE

$$L\frac{dI}{dt} + RI + \frac{Q}{C} = E(t), \tag{B.37}$$

where $LdI/dt$ is the voltage drop across the inductor, $I$ is the current (measured in amperes, A), $L$ is the inductance (measured in henrys, H), $R$ is the resistance (measured in ohms, $\Omega$), $Q$ is the charge on the capacitor (measured in coulombs, C), $C$ is the capacitance (measured in farads, F), $E(t)$ is the time-variable voltage source (measured in volts, V), and $t$ is time (measured in seconds, s). There is a relation

betwen $I$ and $Q$:

$$\frac{dQ}{dt} = I\,. \tag{B.38}$$

Equations (B.37)–(B.38) is a system two ODEs. Solve these for $L = 1$ H, $E(t) = 2\sin\omega t$ V, $\omega^2 = 3.5$ s$^{-2}$, $C = 0.25$ C, $R = 0.2\ \Omega$, $I(0) = 1$ A, and $Q(0) = 1C$. Use the Forward Euler scheme with $\Delta t = 2\pi/(60\omega)$. The solution will, after some time, oscillate with the same period as $E(t)$, a period of $2\pi/\omega$. Simulate 10 periods. (Actually, it turns out that the Forward Euler scheme overestimates the amplitudes of the oscillations. Exercise 9.33 compares the Forward Euler scheme with the more accurate 4th-order Runge-Kutta method.) Name of program file: `electric_circuit.py`. ◇

# A Complete Project

# C

The examples in the ordinary chapters of this book are quite compact and composed to convey programming constructs in a gentle pedagogical way. In this appendix the idea is to solve a more comprehensive real-world problem by programming. The problem solving process gets quite advanced because we bring together elements from physics, mathematics, and programming, in a way that a scientific programmer must master. Each individual element is quite forward in the sense that you have probably met the element already, either in high school physics or mathematics, or in this book. The challenge is to understand the problem, and analyze it by breaking it into a set of simpler elements. It is not necessary to understand this problem solving process in detail. As a computer programmer, all you need to understand is how you translate the given algorithm into a working program and how to test the program. We anticipate that this task should be doable without a thorough understanding of the physics and mathematics of the problem.

You can read the present appendix after the material in the first four chapters are digested. More specifically, you can read Appendices C.1 and C.2 after Chapter 3, while Appendix C.3 requires knowledge about curve plotting from Chapter 4.

Appendix C.1–C.2 can be read after the first three chapters of the book is digested. Appendix C.3 requires the plotting knowledge of Chapter 4.

All Python files associated with this appendix are found in `src/box_spring`.

## C.1 About the Problem: Motion and Forces in Physics

### C.1.1 The Physical Problem

We shall study a simple device which models oscillating systems. A box with mass $m$ and height $b$ is attached to a spring of length $L$ as shown in Figure C.1. The end of the spring is attached to a plate which

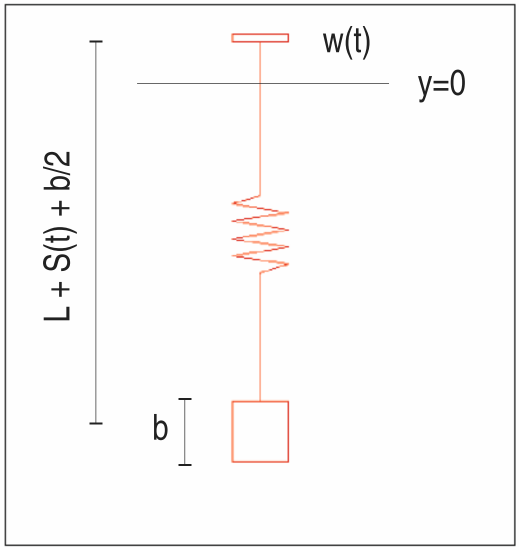



**Fig. C.1** An oscillating system with a box attached to a spring.

we can move up and down with a displacement $w(t)$, where $t$ denotes time. There are two ways the box can be set in motion: we can either stretch or compress the string initially by moving the box up or down, or we can move the plate. If $w = 0$ the box oscillates freely, otherwise we have what is called driven oscillations.

Why will such a system oscillate? When the box moves downward, the spring is stretched, which results in a force that tries to move the box upward. The more we stretch the spring, the bigger the force against the movement becomes. The box eventually stops and starts moving upward with an upward acceleration. At some point the spring is not stretched anymore and there is no spring force on the box, but because of inertia, the box continues its motion upward. This causes the spring to get compressed, causing a force from the spring on the box that acts downward, against the upward movement. The downward force increases in intensity and manages to stop the upward motion. The process repeats itself and results in an oscillatory motion of the box. Since the spring tries to restore the position of the box, we refer to the spring force as a *restoring force*.

You have probably experienced that oscillations in such springs tend to die out with time. There is always a *damping force* that works

against the motion. This damping force may be due to a not perfectly elastic string, and the force can be quite small, but we can also explicitly attach the spring to a damping mechanism to obtain a stronger, controllable damping of the oscillations (as one wants in a car or a mountain bike). We will assume that there is some damping force present in our system, and this can well be a damping mechanism although this is not explicitly included in Figure C.1.

Oscillating systems of the type depicted in Figure C.1 have a huge number of applications throughout science and technology. One simple example is the spring system in a car or bicycle, which you have probably experienced on a bumpy road (the bumps lead to a $w(t)$ function). When your washing machine jumps up and down, it acts as a highly damped oscillating system (and the $w(t)$ function is related to uneven distribution of the mass of the clothes). The pendulum in a wall clock is another oscillating system, not with a spring, but physically the system can (for small oscillations) be modeled as a box attached to a spring because gravity makes a spring-like force on a pendulum (in this case, $w(t) = 0$). Other examples on oscillating systems where this type of equation arise are briefly mentioned in Exercise 9.45. The bottom line is that understanding the dynamics of Figure C.1 is the starting point for understanding the behavior of a wide range of oscillating phenomena in nature and technical devices.

*Goal of the Compuations.* Our aim is to compute the position of the box as a function of time. If we know the position, we can compute the velocity, the acceleration, the spring force, and the damping force. The mathematically difficult thing is to calculate the position – everything else is much easier[1].

We assume that the box moves in the vertical direction only, so we introduce $Y(t)$ as the vertical position of the center point of the box. We shall derive a mathematical equation that has $Y(t)$ as solution. This equation can be solved by an algorithm which can be implemented in a program. Our focus is on the implementation, since this is a book about programming, but for the reader interested in how computers play together with physics and mathematics in science and technology, we also outline how the equation and algorithm arise.

*The Key Quantities.* Let $S$ be the stretch of the spring, where $S > 0$ means stretch and $S < 0$ implies compression. The length of the spring when it is unstretched is $L$, so at a given point of time $t$ the actual length is $L + S(t)$. Given the position of the plate, $w(t)$, the length of

[1] More precisely, to compute the position we must solve a differential equation while the other quantities can be computed by differentiation and simple arithmetics. Solving differential equations is historically considered very difficult, but computers have simplified this task dramatically. Appendix B and Chapters 7.4 and 9.4 are devoted to this topic.

the spring, $L + S(t)$, and the height of the box, $b$, the position $Y(t)$ is then, according to Figure C.1,

$$Y(t) = w(t) - (L + S(t)) - \frac{b}{2}\,. \tag{C.1}$$

You can think as follows: We first "go up" to the plate at $y = w(t)$, then down $L + S(t)$ along the spring and then down $b/2$ to the center of the box. While $L$, $w$, and $b$ must be known as input data, $S(t)$ is unknown and will be output data from the program.

### C.1.2 The Computational Algorithm

Let us now go straight to the programming target and present the recipe for computing $Y(t)$. The algorithm below actually computes $S(t)$, but at any point of time we can easily find $Y(t)$ from (C.1) if we know $S(t)$. The $S(t)$ function is computed at discrete points of time, $t = t_i = i\Delta t$, for $i = 0, 1, \ldots, N$. We introduce the notation $S_i$ for $S(t_i)$. The $S_i$ values can be computed by the following algorithm.

1. Set initial stretch $S_0$ from input data
2. Compute $S_1$ by
$$S_{i+1} = \frac{1}{2m}\left(2mS_i - \Delta t^2\, kS_i + m\left(w_{i+1} - 2w_i + w_{i-1}\right) + \Delta t^2\, mg\right), \tag{C.2}$$
with $i = 0$.
3. For $i = 1, 2, \ldots, N-1$, compute $S_{i+1}$ by
$$\begin{aligned} S_{i+1} = (m+\gamma)^{-1}\ \big(&2mS_i - mS_{i-1} + \gamma\Delta t\, S_{i-1} - \Delta t^2\, kS_i + \\ &m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2\, mg\big)\,. \end{aligned} \tag{C.3}$$

The parameter $\gamma$ equals $\frac{1}{2}\beta\Delta t$. The input data to the algorithm are the mass of the box $m$, a coefficient $k$ characterizing the spring, a coefficient $\beta$ characterizing the amount of damping in the system, the acceleration of gravity $g$, the movement of the plate $w(t)$, the initial stretch of the spring $S_0$, the number of time steps $N$, and the time $\Delta t$ between each computation of $S$ values. The smaller we choose $\Delta t$, the more accurate the computations become.

Now you have two options, either read the derivation of this algorithm in Appendix C.1.3–C.1.4 or jump right to implementation in Appendix C.2.

### C.1.3 Derivation of the Mathematical Model

To derive the algorithm we need to make a mathematical model of the oscillating system. This model is based on physical laws. The most

important physical law for a moving body is Newton's second law of motion:

$$F = ma, \tag{C.4}$$

where $F$ is the sum of all forces on the body, $m$ is the mass of the body, and $a$ is the acceleration of the body. The body here is our box.

Let us first find all the forces on the box. Gravity acts downward with magnitude $mg$. We introduce $F_g = -mg$ as the gravity force, with a minus sign because a negative force acts downward, in negative $y$ direction.

The spring force on the box acts upward if the spring is stretched, i.e., if $S > 0$ we have a positive spring force $F_s$. The size of the force is proportional to the amount of stretching, so we write[2] $F_s = kS$, where $k$ is commonly known as the *spring constant*. We also assume that we have a damping force that is always directed toward the motion and proportional with the "velocity of the stretch", $-dS/dt$. Naming the porportionaly constant $\beta$, we can write the damping force as $F_d = \beta dS/dt$. Note that when $dS/dt > 0$, $S$ increases in time and the box moves downward, the $F_d$ force then acts upward, against the motion, and must be positive. This is the way we can check that the damping force expression has the right sign.

The sum of all forces is now

$$\begin{aligned} F &= F_g + F_s + F_d, \\ &= -mg + kS + \beta \frac{dS}{dt}\,. \end{aligned} \tag{C.5}$$

We now know the left-hand side of (C.4), but $S$ is unknown to us. The acceleration $a$ on the right-hand side of (C.4) is also unknown. However, acceleration is related to movement and the $S$ quantity, and through this relation we can eliminate $a$ as a second unknown. From physics, it is known that the acceleration of a body is the second derivative in time of the position of the body, so in our case,

$$\begin{aligned} a &= \frac{d^2Y}{dt^2}, \\ &= \frac{d^2w}{dt^2} - \frac{d^2S}{dt^2}, \end{aligned} \tag{C.6}$$

(remember that $L$ and $b$ are constant).

Equation (C.4) now reads

$$-mg + kS + \beta \frac{dS}{dt} = m\left(\frac{d^2w}{dt^2} - \frac{d^2S}{dt^2}\right)\,. \tag{C.7}$$

[2] Spring forces are often written in the canonical form "$F = -kx$", where $x$ is the stretch. The reason that we have no minus sign is that our stretch $S$ is positive in the downward (negative) direction.

It is common to collect the unknown terms on the left-hand side and the known quantities on the right-hand side, and let higher-order derivatives appear before lower-order derivatives. With such a reordering of terms we get

$$m\frac{d^2S}{dt^2} + \beta\frac{dS}{dt} + kS = m\frac{d^2w}{dt^2} + mg\,. \tag{C.8}$$

This is the equation governing our physical system. If we solve the equation for $S(t)$, we have the position of the box according to (C.1), the velocity $v$ as

$$v(t) = \frac{dY}{dt} = \frac{dw}{dt} - \frac{dS}{dt}, \tag{C.9}$$

the acceleration as (C.6), and the various forces can be easily obtained from the formulas in (C.5).

A key question is if we can solve (C.8). If $w = 0$, there is in fact a well-known solution which can be written

$$S(t) = \frac{m}{k}g + \begin{cases} e^{-\zeta t}\left(c_1 e^{t\sqrt{\beta^2-1}} + c_2 e^{-t\sqrt{\zeta^2-1}}\right), & \zeta > 1, \\ e^{-\zeta t}(c_1 + c_2 t), & \zeta = 1, \\ e^{-\zeta t}\left[c_1\cos\left(\sqrt{1-\zeta^2}t\right) + c_2\sin\left(\sqrt{1-\zeta^2}t\right)\right], & \zeta < 1\,. \end{cases} \tag{C.10}$$

Here, $\zeta$ is a short form for $\beta/2$, and $c_1$ and $c_2$ are arbitrary constants. That is, the solution (C.10) is not unique.

To make the solution unique, we must determine $c_1$ and $c_2$. This is done by specifying the state of the system at some point of time, say $t = 0$. In the present type of mathematical problem we must specify $S$ and $dS/dt$. We allow the spring to be stretched an amount $S_0$ at $t = 0$. Moreover, we assume that there is no ongoing increase or decrease in the stretch at $t = 0$, which means that $dS/dt = 0$. In view of (C.9), this condition implies that the velocity of the box is that of the plate, and if the latter is at rest, the box is also at rest initially. The conditions at $t = 0$ are called *initial conditions*:

$$S(0) = S_0, \quad \frac{dS}{dt}(0) = 0\,. \tag{C.11}$$

These two conditions provide two equations for the two unknown constants $c_1$ and $c_2$. Without the initial conditions two things happen: (i) there are infinitely many solutions to the problem, and (ii) the computational algorithm in a program cannot start.

Also when $w \neq 0$ one can find solutions $S(t)$ of (C.8) in terms of mathematical expressions, but only for some very specific choices of $w(t)$ functions. With a program we can compute the solution $S(t)$ for any "reasonable" $w(t)$ by a quite simple method. The method gives only an approximate solution, but the approximation can usually be

made as good as desired. This powerful solution method is described below.

### C.1.4 Derivation of the Algorithm

To solve (C.8) on a computer, we do two things:

1. We calculate the solution at some discrete time points $t = t_i = i\Delta t$, $i = 0, 1, 2, \ldots, N$.
2. We replace the derivatives by finite differences, which are approximate expressions for the derivatives.

The first and second derivatives can be approximated by[3]

$$\frac{dS}{dt}(t_i) \approx \frac{S(t_{i+1}) - S(t_{i-1})}{2\Delta t}, \tag{C.12}$$

$$\frac{d^2S}{dt^2}(t_i) \approx \frac{S(t_{i+1}) - 2S(t_i) + S(t_{i-1})}{\Delta t^2}\,. \tag{C.13}$$

It is common to save some writing by introducing $S_i$ as a short form for $S(t_i)$. The formulas then read

$$\frac{dS}{dt}(t_i) \approx \frac{S_{i+1} - S_{i-1}}{2\Delta t}, \tag{C.14}$$

$$\frac{d^2S}{dt^2}(t_i) \approx \frac{S_{i+1} - 2S_i + S_{i-1}}{\Delta t^2}\,. \tag{C.15}$$

Let (C.8) be valid at a point of time $t_i$:

$$m\frac{d^2S}{dt^2}(t_i) + \beta\frac{dS}{dt}(t_i) + kS(t_i) = m\frac{d^2w}{dt^2}(t_i) + mg\,. \tag{C.16}$$

We now insert (C.14) and (C.15) in (C.16) (observe that we can approximate $d^2w/dt^2$ in the same way as we approximate $d^2S/dt^2$):

$$m\frac{S_{i+1} - 2S_i + S_{i-1}}{\Delta t^2} + \beta\frac{S_{i+1} - S_{i-1}}{2\Delta t} + kS_i = m\frac{w_{i+1} - 2w_i + w_{i-1}}{\Delta t^2} + mg\,. \tag{C.17}$$

The computational algorithm starts with knowing $S_0$, then $S_1$ is computed, then $S_2$, and so on. Therefore, in (C.17) we can assume that $S_i$ and $S_{i-1}$ are already computed, and that $S_{i+1}$ is the new unknown to calculate. Let us as usual put the unknown terms on the left-hand side (and multiply by $\Delta t^2$):

$$\begin{aligned} mS_{i+1} + \gamma S_{i+1} = 2mS_i - mS_{i-1} + \gamma S_{i-1} - \Delta t^2\, kS_i + \\ m\,(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2\, mg, \end{aligned} \tag{C.18}$$

[3] See Appendices A and B for derivations of such formulas.

where we have introduced the short form $\gamma = \frac{1}{2}\beta\Delta t$ to save space. Equation (C.18) can easily be solved for $S_{i+1}$:

$$S_{i+1} = (m+\gamma)^{-1}\big(\, 2mS_i - mS_{i-1} + \gamma\Delta t\, S_{i-1} - \Delta t^2\, kS_i + m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2\, mg\big), \tag{C.19}$$

One fundamental problem arises when we try to start the computations. We know $S_0$ and want to apply (C.19) for $i = 0$ to calculate $S_1$. However, (C.19) involves $S_{i-1}$, that is, $S_{-1}$, which is an unknown value at a point of time *before* we compute the motion. The initial conditions come to rescue here. Since $dS/dt = 0$ at $t = 0$ (or $i = 0$), we can approximate this condition as

$$\frac{S_1 - S_{-1}}{2\Delta t} = 0 \quad : \quad S_{-1} = S_1\,. \tag{C.20}$$

Inserting this relation in (C.19) when $i = 0$ gives a special formula for $S_1$ (or $S_{i+1}$ with $i = 0$, if we want):

$$S_{i+1} = \frac{1}{2m}\left(2mS_i - \Delta t^2\, kS_i + m\,(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2\, mg\right)\,. \tag{C.21}$$

Remember that $i = 0$ in this formula. The overall algorithm is summarized below:

1. Initialize $S_0$ from initial condition
2. Use (C.21) to compute $S_{i+1}$ for $i = 0$
3. For $i = 0, 1, 2, \ldots, N-1$, use (C.19) to compute $S_{i+1}$

## C.2 Program Development and Testing

### C.2.1 Implementation

The aim now is to implement the algorithm on page 628 in a Python program. There are naturally two parts of the program, one where we read input data such as $L$, $m$, and $w(t)$, and one part where we run the computational algorithm. Let us write a function for each part.

The set of input data to the program consists of the mathematical symbols

- $m$ (the mass of the box)
- $b$ (the height of the box)
- $L$ (the length of the unstretched spring)
- $\beta$ (coefficient for the damping force)
- $k$ (coefficient for the spring force)
- $\Delta t$ (the time step between each $S_i$ calculation)
- $N$ (the number of computed time steps)
- $S_0$ (the initial stretch of the spring)

- $w(t)$ (the vertical displacement of the plate)
- $g$ (acceleration of gravity)

We make a function `init_prms` for initializing these input parameters from option-value pairs on the command line. That is, the user provides pairs like `-m 2` and `-dt 0.1` (for $\Delta t$). The `getopt` module from Chapter 3.2.4 can be used for this purpose. We supply default values for all parameters as arguments to the `init_prms` function. The function returns all these parameters with the changes that the user has specified on the command line. The $w$ parameter is given as a string expression (called `w_formula` below), and the `StringFunction` tool from Chapter 3.1.4 can be used to turn the formula into a working Python function. An algorithmic sketch of the tasks in the `init_prms` function can be expressed by some pseudo Python code:

```
def init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N):
    import getopt, sys
    try:
        options, args = getopt.getopt(
            sys.argv[1:], '',
            <list of legal command-line options>)
    except getopt.GetoptError, e:
        <handle error in command-line options>

    for option, value in options:
        if option in ('--m', '--mass'):
            m = float(value)
        elif option in ('--b', '--boxheight'):
            b = float(value)
        elif ... # treat all other parameters

    from scitools.StringFunction import StringFunction
    w = StringFunction(w_formula, independent_variables='t')
    return m, b, L, k, beta, S0, dt, g, w, N
```

With such a sketch as a start, we can complete the indicated code and arrive at a working function for specifying input parameters to the mathematical model:

```
def init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N):
    import getopt, sys
    try:
        options, args = getopt.getopt(
            sys.argv[1:], '',
            ['m=', 'mass=',
             'b=', 'boxheight=',
             'L=', 'spring-length=',
             'k=', 'spring-stiffness=',
             'beta=', 'spring-damping=',
             'S0=', 'initial-position=',
             'dt=','timestep=',
             'g=', 'gravity=',
             'w=',
             'N='])
    except getopt.GetoptError, e:
        print 'Error in command-line option:\n', e
        sys.exit(1)

    for option, value in options:
        if option in ('--m', '--mass'):
```

```
            m = float(value)
        elif option in ('--b', '--boxheight'):
            b = float(value)
        elif option in ('--L', '--spring-length'):
            L = float(value)
        elif option in ('--k', '--spring-stiffness'):
            k = float(value)
        elif option in ('--beta', '--spring-damping'):
            beta = float(value)
        elif option in ('--S0', '--initial-position'):
            S0 = float(value)
        elif option in ('--dt', '--timestep'):
            dt = float(value)
        elif option in ('--g', '--gravity'):
            g = float(value)
        elif option in ('--w',):
            w_formula = value  # string
        elif option == '--N':
            N = int(value)

    from scitools.StringFunction import StringFunction
    w = StringFunction(w_formula, independent_variables='t')
    return m, b, L, k, beta, S0, dt, g, w, N
```

You may wonder why we specify $g$ (gravity) since this is a known constant, but it is useful to turn off the gravity force to test the program. Just imagine the oscillations take place in the horizontal direction – the mathematical model is the same, but $F_g = 0$, which we can obtain in our program by setting the input parameter `g` to zero.

The computational algorithm is quite easy to implement, as there is a quite direct translation of the mathematical algorithm in Appendix C.1.2 to valid Python code. The $S_i$ values can be stored in a list or array with indices going from 0 to $N$. To allow readers to follow the code here without yet having digested Chapter 4, we use a plain list. The function for computing $S_i$ reads

```
def solve(m, k, beta, S0, dt, g, w, N):
    S = [0.0]*(N+1)      # output list
    gamma = beta*dt/2.0  # short form
    t = 0
    S[0] = S0
    # special formula for first time step:
    i = 0
    S[i+1] = (1/(2.0*m))*(2*m*S[i] - dt**2*k*S[i] +
             m*(w(t+dt) - 2*w(t) + w(t-dt)) + dt**2*m*g)
    t = dt

    for i in range(1,N):
        S[i+1] = (1/(m + gamma))*(2*m*S[i] - m*S[i-1] +
                                  gamma*dt*S[i-1] - dt**2*k*S[i] +
                                  m*(w(t+dt) - 2*w(t) + w(t-dt))
                                  + dt**2*m*g)
        t += dt
    return S
```

The primary challenge in coding the algorithm is to set the index `t` and the time `t` right. Recall that in the updating formula for `S[i+1]` at time `t+dt`, the time on the right-hand side shall be the time at time

step i, so the t+=dt update must come after S[i+1] is computed. The same is important in the special formula for the first time step as well.

A main program will typically first set some default values of the 10 input parameters, then call init_prms to let the user adjust the default values, and then call solve to compute the $S_i$ values:

```
# default values:
from math import pi
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 80;

m, b, L, k, beta, S0, dt, g, w, N = \
   init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)
S = solve(m, k, beta, S0, dt, g, w, N)
```

So, what shall we do with the solution S? We can write out the values of this list, but the numbers do not give an immediate feeling for how the box moves. It will be better to graphically illustrate the $S(t)$ function, or even better, the $Y(t)$ function. This is straightforward with the techniques from Chapter 4 and is treated in Appendix C.3. In Chapter 9.5, we develop a drawing tool for drawing figures like Figure C.1. By drawing the box, string, and plate at every time level we compute $S_i$, we can use this tool to make a moving figure that illustrates the dynamics of the oscillations. Already now you can play around with a program doing that (box_spring_figure_anim.py).

### C.2.2 Callback Functionality

It would be nice to make some graphics of the system while the computations take place, not only after the S list is ready. The user must then put some relevant statements in between the statements in the algorithm. However, such modifications will depend on what type of analysis the user wants to do. It is a bad idea to mix user-specific statements with general statements in a general algorithm. We therefore let the user provide a function that the algorithm can call after each $S_i$ value is computed. This is commonly called a *callback* function (because a general function calls back to the user's program to do a user-specific task). To this callback function we send three key quantities: the S list, the point of time ($t$), and the time step number ($i+1$), so that the user's code gets access to these important data.

If we just want to print the solution to the screen, the callback function can be as simple as

```
def print_S(S, t, step):
    print 't=%.2f  S[%d]=%+g' % (t, step, S[step])
```

In the solve function we take the callback function as a keyword argument user_action. The default value can be an empty function, which we can define separately:

```
def empty_func(S, time, time_step_no):
    return None

def solve(m, k, beta, S0, dt, g, w, N,
          user_action=empty_func):
    ...
```

However, it is quicker to just use a lambda function (see Chapter 2.2.11):

```
def solve(m, k, beta, S0, dt, g, w, N,
          user_action=lambda S, time, time_step_no: None):
```

The new `solve` function has a call to `user_action` each time a new $S$ value has been computed:

```
def solve(m, k, beta, S0, dt, g, w, N,
          user_action=lambda S, time, time_step_no: None):
    """Calculate N steps forward. Return list S."""
    S = [0.0]*(N+1)      # output list
    gamma = beta*dt/2.0  # short form
    t = 0
    S[0] = S0
    user_action(S, t, 0)
    # special formula for first time step:
    i = 0
    S[i+1] = (1/(2.0*m))*(2*m*S[i] - dt**2*k*S[i] +
               m*(w(t+dt) - 2*w(t) + w(t-dt)) + dt**2*m*g)
    t = dt
    user_action(S, t, i+1)

    # time loop:
    for i in range(1,N):
        S[i+1] = (1/(m + gamma))*(2*m*S[i] - m*S[i-1] +
                                  gamma*dt*S[i-1] - dt**2*k*S[i] +
                                  m*(w(t+dt) - 2*w(t) + w(t-dt))
                                  + dt**2*m*g)
        t += dt
        user_action(S, t, i+1)

    return S
```

The two last arguments to `user_action` must be carefully set: these should be time value and index for the most recently computed `S` value.

### C.2.3 Making a Module

The `init_prms` and `solve` functions can now be combined with many different types of main programs and `user_action` functions. It is therefore preferable to have the general `init_prms` and `solve` functions in a module `box_spring` and import these functions in more user-specific programs. Making a module out of `init_prms` and `solve` is, according to Chapter 3.5, quite trivial as we just need to put the functions in a file `box_spring.py`.

It is always a good habit to include a test block in module files. To make the test block small, we place the statements in a separate

function `_test` and just call `_test` in the test block. The initial underscore in the name `_test` prevents this function from being imported by a `from box_spring import *` statement. Our test here simply prints solution at each time level. The following code snippet is then added to the module file to include a test block:

```
def _test():
    def print_S(S, t, step):
        print 't=%.2f  S[%d]=%+g' % (t, step, S[step])

    # default values:
    from math import pi
    m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
    dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 80;

    m, b, L, k, beta, S0, dt, g, w, N = \
       init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)
    S = solve(m, k, beta, S0, dt, g, w, N,
              user_action=print_S)

if __name__ == '__main__':
    _test()
```

### C.2.4 Verification

To check that the program works correctly, we need a series of problems where the solution is known. These test cases must be specified by someone with a good physical and mathematical understanding of the problem being solved. We already have a solution formula (C.10) that we can compare the computations with, and more tests can be made in the case $w \neq 0$ as well.

However, before we even think of checking the program against the formula (C.10), we should perform some much simpler tests. The simplest test is to see what happens if we do nothing with the system. This solution is of course not very exciting – the box is at rest, but it is in fact exciting to see if our program reproduces the boring solution. Many bugs in the program can be found this way! So, let us run the program `box_spring.py` with `-S0 0` as the only command-line argument. The output reads

```
t=0.00  S[0]=+0
t=0.16  S[1]=+0.121026
t=0.31  S[2]=+0.481118
t=0.47  S[3]=+1.07139
t=0.63  S[4]=+1.87728
t=0.79  S[5]=+2.8789
t=0.94  S[6]=+4.05154
t=1.10  S[7]=+5.36626
...
```

Something happens! All `S[1]`, `S[2]`, and so forth should be zero. What is the error?

There are two directions to follow now: we can either visualize the solution to understand more of what the computed $S(t)$ function looks

like (perhaps this explains what is wrong), or we can dive into the algorithm and compute `S[1]` by hand to understand why it does not become zero. Let us follow both paths.

First we print out all terms on the right-hand side of the statement that computes `S[1]`. All terms except the last one ($\Delta t^2\, mg$) are zero. The gravity term causes the spring to be stretched downward, which causes oscillations. We can see this from the governing equation (C.8) too: If there is no motion, $S(t) = 0$, the derivatives are zero (and $w = 0$ is default in the program), and then we are left with

$$kS = mg \quad : \quad S = \frac{m}{k} g\,. \tag{C.22}$$

This result means that if the box is at rest, the spring is stretched (which is reasonable!). Either we have to start with $S(0) = \frac{m}{k} g$ in the equilibrium position, or we have to turn off the gravity force by setting `-g 0` on the command line. Setting either `-S0 0 -g 0` or `-S0 9.81` shows that the whole `S` list contains either zeros or 9.81 values (recall that $m = k = 1$ so $S_0 = g$). This constant solution is correct, and the coding looks promising.

We can also plot the solution using the program `box_spring_plot`:

Terminal

```
box_spring_plot.py --S0 0 --N 200
```

Figure C.2 shows the function $Y(t)$ for this case where the initial stretch is zero, but gravity is causing a motion. With some mathematical analysis of this problem we can establish that the solution is correct. We have that $m = k = 1$ and $w = \beta = 0$, which implies that the governing equation is

$$\frac{d^2S}{dt^2} + S = g, \quad S(0) = 0,\ dS/dt(0) = 0\,.$$

Without the $g$ term this equation is simple enough to be solved by basic techniques you can find in most introductory books on differential equations. Let us therefore get rid of the $g$ term by a little trick: we introduce a new variable $T = S - g$, and by inserting $S = T + g$ in the equation, the $g$ is gone:

$$\frac{d^2T}{dt^2} + T = 0, \quad T(0) = -g,\ \frac{dT}{dt}(0) = 0\,. \tag{C.23}$$

This equation is of a very well-known type and the solution reads $T(t) = -g\cos t$, which means that $S(t) = g(1 - \cos t)$ and

$$Y(t) = -L - g(1 - \cos t) - \frac{b}{2}\,.$$

With $L = 10$, $g \approx 10$, and $b = 2$ we get oscillations around $y \approx 21$ with a period of $2\pi$ and a start value $Y(0) = -L - b/2 = 11$. A rough visual inspection of the plot shows that this looks right. A more thorough analysis would be to make a test of the numerical values in a new callback function (the program is found in `box_spring_test1.py`):

```
from box_spring import init_prms, solve
from math import cos

def exact_S_solution(t):
    return g*(1 - cos(t))

def check_S(S, t, step):
    error = exact_S_solution(t) - S[step]
    print 't=%.2f  S[%d]=%+g error=%g' % (t, step, S[step], error)

# fixed values for a test:
from math import pi
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 0
dt = 2*pi/40; g = 9.81; N = 200

def w(t):
    return 0

S = solve(m, k, beta, S0, dt, g, w, N, user_action=check_S)
```

The output from this program shows increasing errors with time, up as large values as 0.3. The difficulty is to judge whether this is the error one must expect because the program computes an approximate solution, or if this error points to a bug in the program – or a wrong mathematical formula.

From these sessions on program testing you will probably realize that verification of mathematical software is challenging. In particular, the design of the test problems and the interpretation of the numerical output require quite some experience with the interplay between physics (or another application discipline), mathematics, and programming.

## C.3 Visualization

The purpose of this section is to add graphics to the oscillating system application developed in Appendix C.2. Recall that the function `solve` solves the problem and returns a list `S` with indices from `0` to `N`. Our aim is to plot this list and various physical quantities computed from it.

### C.3.1 Simultaneous Computation and Plotting

The `solve` function makes a call back to the user's code through a callback function (the `user_action` argument to `solve`) at each time level. The callback function has three arguments: `S`, the time, and the

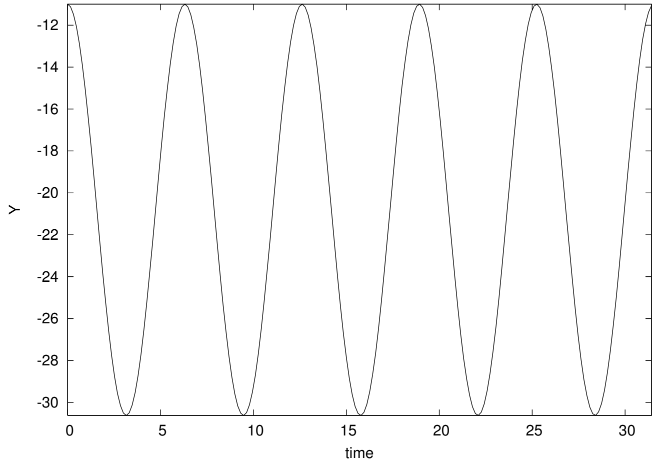



**Fig. C.2** Positions $Y(t)$ of an oscillating box with $m = k = 1$, $w = \beta = 0$, $g = 9.81$, $L = 10$, and $b = 2$.

current time step number. Now we want the callback function to plot the position $Y(t)$ of the box during the computations. In principle this is easy, but `S` is longer than we want to plot, because `S` is allocated for the whole time simulation while the `user_action` function is called at time levels where only the indices in `S` up to the current time level have been computed (the rest of the elements in `S` are zero). We must therefore use a sublist of `S`, from time zero and up to the current time. The callback function we send to `solve` as the `user_action` argument can then be written like this:

```
def plot_S(S, t, step):
    if step == 0:        # nothing to plot yet
        return None

    tcoor = linspace(0, t, step+1)
    S = array(S[:len(tcoor)])
    Y = w(tcoor) - L - S - b/2.
    plot(tcoor, Y)
```

Note that `L`, `dt`, `b`, and `w` must be global variables in the user's main program.

The major problem with the `plot_S` function shown is that the `w(tcoor)` evaluation does not work. The reason is that `w` is a `StringFunction` object, and according to Chapter 4.4.3, `StringFunction` objects do not work with array arguments unless we call their `vectorize` function once. We therefore need to do a

```
w.vectorize(globals())
```

before calling `solve` (which calls `plot_S` repeatedly). Here is the main program with this important statement:

```
from box_spring import init_prms, solve
from scitools.std import *

# default values:
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 200;

m, b, L, k, beta, S0, dt, g, w, N = \
   init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)

w.vectorize(globals())

S = solve(m, k, beta, S0, dt, g, w, N, user_action=plot_S)
```

Now the `plot_S` function works fine. You can try the program out by running

Terminal

```
box_spring_plot_v1.py
```

*Fixing Axes.* Both the $t$ and the $y$ axes adapt to the solution array in every plot. The adaptation of the $y$ is okay since it is difficult to predict the future minimum and maximum values of the solution, and hence it is most natural to just adapt the $y$ axis to the computed $Y$ points so far in the simulation. However, the $t$ axis should be fixed throughout the simulation, and this is easy since we know the start and end times. The relevant `plot` call now becomes[4]

```
plot(tcoor, Y,
     axis=[0, N*dt, min(Y), max(Y)],
     xlabel='time', ylabel='Y')
```

At the end of the simulation it can be nice to make a hardcopy of the last `plot` command performed in the `plot_S` function. We then just call

```
hardcopy('tmp_Y.eps')
```

after the `solve` function is finished.

In the beginning of the simulation it is convenient to skip plotting for a number of steps until there are some interesting points to visualize and to use for computing the axis extent. We also suggest to apply the recipe at the end of Chapter 4.4.3 to vectorize `w`. More precisely, we use `w.vectorize` in general, but turn to NumPy's `vectorize` feature only if the string formula contains an inline `if-else` test (to avoid requiring users to use `where` to vectorize the string expressions). One reason for paying attention to `if-else` tests in the $w$ formula is that sudden movements of the plate are of interest, and this gives rise to step functions and strings like `'1 if t>0 else 0'`. A main program with all these features is listed next.

[4] Note that the final time is $T = N\Delta t$.

```
from box_spring import init_prms, solve
from scitools.std import *

def plot_S(S, t, step):
    first_plot_step = 10            # skip the first steps
    if step < first_plot_step:
        return

    tcoor = linspace(0, t, step+1)  # t = dt*step
    S = array(S[:len(tcoor)])
    Y = w(tcoor) - L - S - b/2.0    # (w, L, b are global vars.)

    plot(tcoor, Y,
         axis=[0, N*dt, min(Y), max(Y)],
         xlabel='time', ylabel='Y')


# default values:
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 200

m, b, L, k, beta, S0, dt, g, w, N = \
   init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)

# vectorize the StringFunction w:
w_formula = str(w)  # keep this to see if w=0 later
if ' else ' in w_formula:
    w = vectorize(w)        # general vectorization
else:
    w.vectorize(globals())  # more efficient (when no if)

S = solve(m, k, beta, S0, dt, g, w, N, user_action=plot_S)

# first make a hardcopy of the the last plot of Y:
hardcopy('tmp_Y.eps')
```

### C.3.2 Some Applications

What if we suddenly, right after $t = 0$, move the plate upward from $y = 0$ to $y = 1$? This will set the system in motion, and the task is to find out what the motion looks like.

There is no initial stretch in the spring, so the initial condition becomes $S_0 = 0$. We turn off gravity for simplicity and try a $w = 1$ function since the plate has the position $y = w = 1$ for $t > 0$:

Terminal

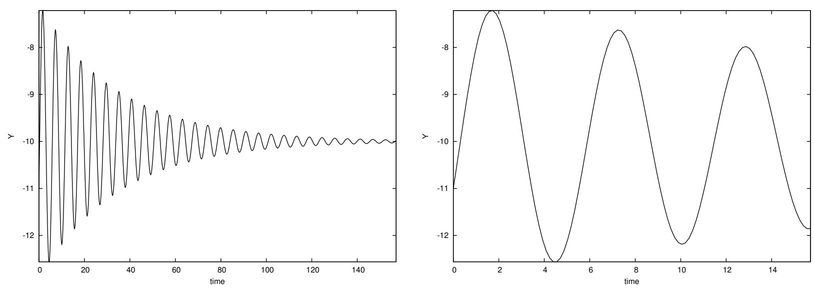
```
box_spring_plot.py --w '1' --S 0 --g 0
```

Nothing happens. The reason is that we specify $w(t) = 1$, but in the equation only $d^2w/dt^2$ has an effect and this quantity is zero. What we need to specify is a step function: $w = 0$ for $t \leq 0$ and $w = 1$ for $t > 0$. In Python such a function can be specified as a string expression `'1 if t>0 else 0'`. With a step function we obtain the right initial jump of the plate:

Terminal

```
box_spring_plot.py --w '1 if t > 0 else 0' \
        --S0 0 --g 0 --N 1000 --beta 0.1
```

Figure C.3 displays the solution. We see that the damping parameter has the effect of reducing the amplitude of $Y(t)$, and the reduction looks exponential, which is in accordance with the exact solution (C.10) (although this formula is not valid in the present case because $w \neq 0$ – but one gets the same exponential reduction even in this case). The box is initially located in $Y = 0-(10+0)-2/2 = -11$. During the first time step we get a stretch $S = 0.5$ and the plate jumps up to $y = 1$ so the box jumps to $Y = 1-(10+0.5)-2/2 = -10.5$. In Figure C.3b we that the box starts correctly out and jumps upwards, as expected.




(a) (b)

**Fig. C.3** Plot of the position of an oscillating box where the end point of the spring ($w(t)$) is given a sudden movement at $t = 0$. Other parameters are $m = k = 1$, $\beta = 0.1$, $g = 0$, $S_0 = 0$. (a) 1000 time steps; (b) 100 steps for magnifying the first oscillation cycles.

More exciting motions of the box can be obtained by moving the plate back and forth in time, see for instance Figure C.4 on page 645.

### C.3.3 Remark on Choosing $\Delta t$

If you run the `box_spring_plot.py` program with a large `-dt` argument (for $\Delta t$), strange things may happen. Try `-dt 2 -N 20` as command-line arguments and observe that $Y$ jumps up and down in a saw tooth fashion so we clearly have too large time steps. Then try `-dt 2.1 -N 20` and observe that $Y$ takes on very large values ($10^5$). This highly non-physical result points to an error in the program. However, the problem is not in the program, but in the numerical method used to solve (C.8). This method becomes unstable and hence useless if $\Delta t$ is larger than a critical value. We shall not dig further into such problems, but just notice that mathematical models on a computer must be used with care, and that a serious user of simulation programs must understand how the mathematical methods work in detail and what their limitations are.

### C.3.4 Comparing Several Quantities in Subplots

So far we have plotted $Y$, but there are other interesting quantities to look at, e.g., $S$, $w$, the spring force, and the damping force. The spring force and $S$ are proportional, so only one of these is necessary to plot. Also, the damping force is relevant only if $\beta \neq 0$, and $w$ is only relevant if the string formula is different from the default value `'0'`.

All the mentioned additional plots can be placed in the same figure for comparison. To this end, we apply the `subfigure` command in Easyviz and create a row of individual plots. How many plots we have depends on the values of `str(w)` and `beta`. The relevant code snippet for creating the additional plots is given below and appears after the part of the main program shown above.

```
# make plots of several additional interesting quantities:
tcoor = linspace(0, tstop, N+1)
S = array(S)

plots = 2         # number of rows of plots
if beta != 0:
    plots += 1
if w_formula != '0':
    plots += 1

# position Y(t):
plot_row = 1
subplot(plots, 1, plot_row)
Y = w(tcoor) - L - S - b/2.0
plot(tcoor, Y, xlabel='time', ylabel='Y')

# spring force (and S):
plot_row += 1
subplot(plots, 1, plot_row)
Fs = k*S
plot(tcoor, Fs, xlabel='time', ylabel='spring force')

if beta != 0:
    plot_row += 1
    subplot(plots, 1, plot_row)
    Fd = beta*diff(S)  # diff is in numpy
    # len(diff(S)) = len(S)-1 so we use tcoor[:-1]:
    plot(tcoor[:-1], Fd, xlabel='time', ylabel='damping force')

if w_formula != '0':
    plot_row += 1
    subplot(plots, 1, plot_row)
    w_array = w(tcoor)
    plot(tcoor, w_array, xlabel='time', ylabel='w(t)')

# save this multi-axis plot in a file:
hardcopy('tmp.eps')
```

Figure C.4 displays what the resulting plot looks like for a test case with an oscillating plate ($w$). The command for this run is

Terminal

```
box_spring_plot.py --S0 0 --w '2*(cos(8*t)-1)' \
                   --N 600 --dt 0.05236
```

The rapid oscillations of the plate require us to use a smaller $\Delta t$ and more steps (larger $N$).

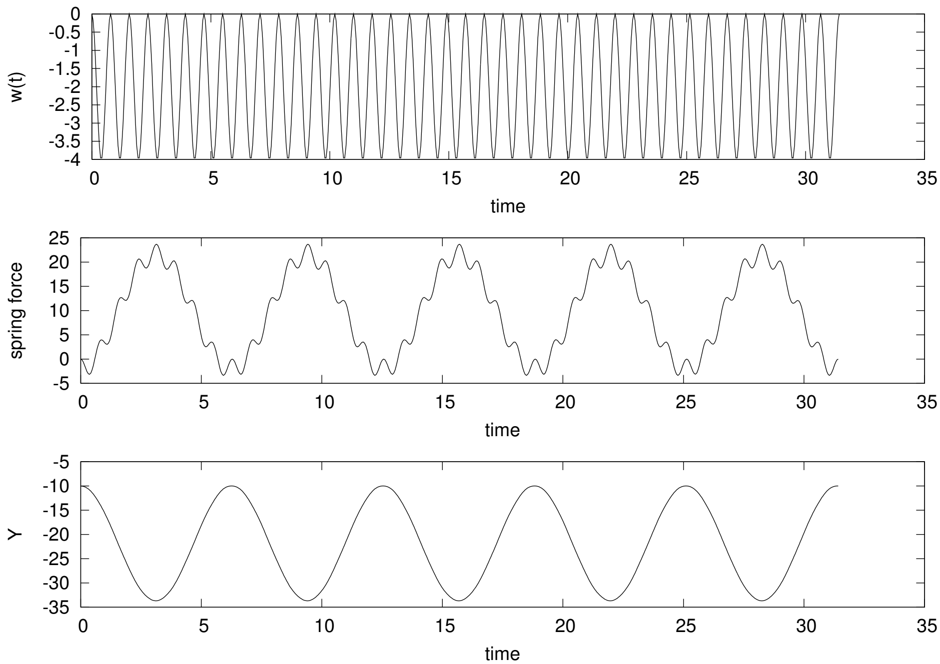



**Fig. C.4** Plot of the plate position $w(t)$, the spring force (proportional to $S(t)$), and the position $Y(t)$ for a test problem where $w(t) = 2(\cos(8t) - 1)$, $\beta = g = 0$, $m = k = 1$, $S_0 = 0$, $\Delta t = 0.5236$, and $N = 600$.

### C.3.5 Comparing Approximate and Exact Solutions

To illustrate multiple curves in the same plot and animations we turn to a slightly different program. The task now is to visually investigate how the accuracy of the computations depends on the $\Delta t$ parameter. The smaller $\Delta t$ is, the more accurate the solution $S$ is. To look into this topic, we need a test problem with known solution. Setting $m = k = 1$ and $w = 0 = \beta = 0$ implies the exact solution $S(t) = g(1 - \cos t)$ (see Appendix C.2.4). The `box_spring_test1.py` program from Appendix C.2.4 can easily be extended to plot the calculated solution together with the exact solution. We drop the `user_action` callback function and just make the plot after having the complete solution `S` returned from the `solve` function:

```
tcoor = linspace(0, N*dt, len(S))
exact = exact_S_solution(tcoor)
plot(tcoor, S, 'r', tcoor, exact, 'b',
     xlabel='time', ylabel='S',
     legend=('computed S(t)', 'exact S(t)'),
     hardcopy='tmp_S.eps')
```

The two curves tend to lie upon each other, so to get some more insight into the details of the error, we plot the error itself, in a separate plot window:

```
figure()      # new plot window
S = array(S)  # turn list into NumPy array for computations
error = exact - S
plot(tcoor, error, xlabel='time', ylabel='error',
     hardcopy='tmp_error.eps')
```

The error increases in time as the plot in Figure C.5a clearly shows.

### C.3.6 Evolution of the Error as $\Delta t$ Decreases

Finally, we want to investigate how the error curve evolves as the time step $\Delta t$ decreases. In a loop we halve $\Delta t$ in each pass, solve the problem, compute the error, and plot the error curve. From the finite difference formulas involved in the computational algorithm, we can expect that the error is of order $\Delta t^2$. That is, if $\Delta t$ is halved, the error should be reduced by 1/4.

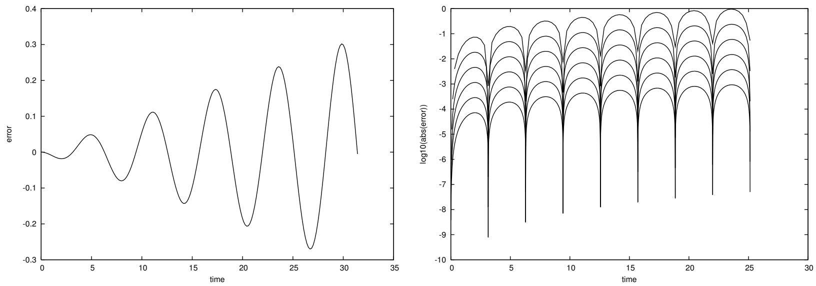
The resulting plot of error curves is not very informative because the error reduces too quickly (by several orders of magnitude). A better plot is obtained by taking the logarithm of the error. Since an error curve may contain positive and negative elements, we take the absolute value of the error before taking the logarithm. We also note that $S_0$ is always correct, so it is necessary to leave out the initial value of the error array to avoid the logarithm of zero.

The ideas of the previous two paragraphs can be summarized in a Python code snippet:

```
figure()      # new plot window
dt = 2*pi/10
tstop = 8*pi  # 4 periods
N = int(tstop/dt)
for i in range(6):
    dt /= 2.0
    N *= 2
    S = solve(m, k, beta, S0, dt, g, w, N)
    S = array(S)
    tcoor = linspace(0, tstop, len(S))
    exact = exact_S_solution(tcoor)
    abserror = abs(exact - S)
    # drop abserror[0] since it is always zero and causes
    # problems for the log function:
    logerror = log10(abserror[1:])
    plot(tcoor[1:], logerror, 'r', xlabel='time',
         ylabel='log10(abs(error))')
    hold('on')
hardcopy('tmp_errors.eps')
```

The resulting plot is shown in Figure C.5b.

Visually, it seems to be a constant distance between the curves in Figure C.5b. Let $d$ denote this difference and let $E_i$ be the absolute




(a) (b)

**Fig. C.5** Error plots for a test problem involving an oscillating system: (a) the error as a function of time; (b) the logarithm of the absolute value of the error as a function of time, where $\Delta t$ is reduced by one half from one curve to the next one below.

error curve associated with $\Delta t$ in the $i$-th pass in the loop. What we plot is $\log_{10} E_i$. The difference between two curves is then $D_{i+1} = \log_{10} E_i - \log_{10} E_{i+1} = \log_{10}(E_i/E_{i+1})$. If this difference is roughly 0.5 as we see from Figure C.5b, we have

$$\log_{10} \frac{E_i}{E_{i+1}} = d = 0.5 \quad : \quad E_{i+1} = \frac{1}{3.16} E_i \, .$$

That is, the error is reduced, but not by the theoretically expected factor 4. Let us investigate this topic in more detail by plotting $D_{i+1}$.

We make a loop as in the last code snippet, but store the `logerror` array from the previous pass in the loop ($E_i$) in a variable `logerror_prev` such that we can compute the difference $D_{i+1}$ as

```
logerror_diff = logerror_prev - logerror
```

There are two problems to be aware of now in this array subtraction: (i) the `logerror_prev` array is not defined before the second pass in the loop (when `i` is one or greater), and (ii) `logerror_prev` and `logerror` have different lengths since `logerror` has twice as many time intervals as `logerror_prev`. Numerical Python does not know how to compute this difference unless the arrays have the same length. We therefore need to use every two elements in `logerror`:

```
logerror_diff = logerror_prev - logerror[::2]
```

An additional problem now arises because the set of time coordinates, `tcoor`, in the current pass of the loop also has twice as many intervals so we need to plot `logerror_diff` against `tcoor[::2]`.

The complete code snippet for plotting differences between the logarithm of the absolute value of the errors now becomes

```
figure()
dt = 2*pi/10
tstop = 8*pi  # 4 periods
```

```
N = int(tstop/dt)
for i in range(6):
    dt /= 2.0
    N *= 2
    S = solve(m, k, beta, S0, dt, g, w, N)
    S = array(S)
    tcoor = linspace(0, tstop, len(S))
    exact = exact_S_solution(tcoor)
    abserror = abs(exact - S)
    logerror = log10(abserror[1:])
    if i > 0:
        logerror_diff = logerror_prev - logerror[::2]
        plot(tcoor[1::2], logerror_diff, 'r', xlabel='time',
             ylabel='difference in log10(abs(error))')
        hold('on')
        meandiff = mean(logerror_diff)
        print 'average log10(abs(error)) difference:', meandiff
    logerror_prev = logerror
hardcopy('tmp_errors_diff.eps')
```

Figure C.6 shows the result. We clearly see that the differences between

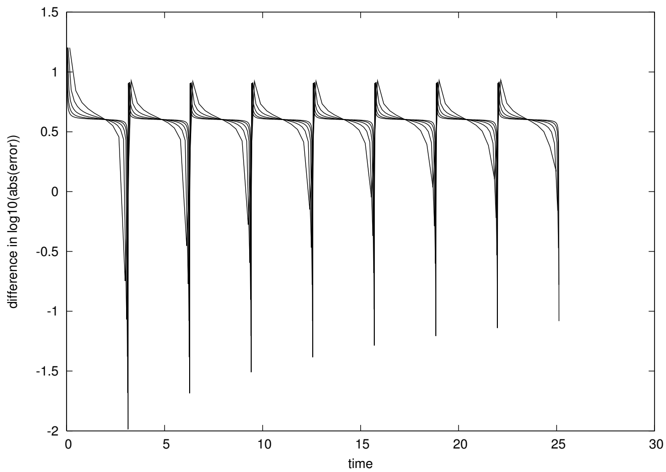



**Fig. C.6** Differences between the curves in Figure C.5b.

the curves in Figure C.5b are almost the same even if $\Delta t$ is reduced by several orders of magnitude.

In the loop we also print out the average value of the difference curves in Figure C.6:

```
average log10(abs(error)) difference: 0.558702094666
average log10(abs(error)) difference: 0.56541814902
average log10(abs(error)) difference: 0.576489014172
average log10(abs(error)) difference: 0.585704362507
average log10(abs(error)) difference: 0.592109360025
```

These values are "quite constant". Let us use 0.57 as an representative value and see what it implies. Roughly speaking, we can then say that

$$\log_{10} E_i - \log_{10} E_{i+1} = 0.57 .$$

Collecting the two first terms and applying the exponential function $10^x$ on both sides we get that

$$E_{i+1} = \frac{1}{3.7} E_i \,.$$

This error reduction when $\Delta t$ is decreased is not quite as good as we would theoretically expect (1/4), but it is close. The purpose of this brief analysis is primarily to show how errors can be explored by plotting, and how we can take advantage of array computing to produce various quantites of interest in a problem. A more thorough investigation of how the error depends on $\Delta t$ would use time integrals of the error instead of the complete error curves.

Again we mention that the complete problem analyzed in this appendix is challenging to understand because of its mix of physics, mathematics, and programming. In real life, however, problem solving in science and industry involve multi-disciplinary projects where people with different competence work together. As a scientific programmer you must then be able to fully understand what to program and how to verify the results. This is a requirement in the current summarizing example too. You have to accept that your programming problem is buried in a lot of physical and mathematical details.

Having said this, we expect that most readers of this book also gain a background in physics and mathematics so that the present summarizing example can be understood in complete detail, at least at some later stage.

## C.4 Exercises

**Exercise C.1.** *Use a w function with a step.*

Set up a problem with the `box_spring_plot.py` program where the initial stretch in the spring is 1 and there is no gravity force. Between $t = 20$ and $t = 30$ we move the plate suddenly from 0 to 2 and back again:

$$w(t) = \begin{cases} 2, \, 20 < t < 30, \\ 0, \, otherwise \end{cases}$$

Run this problem and view the solution. ◇

**Exercise C.2.** *Make a callback function in Exercise C.1.*

Doing Exercise C.1 shows that the $Y$ position increases significantly in magnutude when the “jump” the plate upward and back again at $t = 20$ and $t = 30$, respectively. Make a program where you import from the `box_spring` module and provide a callback function that checks if $Y < 9$ and then aborts the program. Name of program file: `box_spring_Ycrit.py`. ◇

**Exercise C.3.** *Improve input to the simulation program.*

The oscillating system in Appendix C.1 has an equilibrium position $S = mg/k$, see (C.22) on page 638. A natural case is to let the box start at rest in this position and move the plate to induce osciallations. We must then prescribe $S_0 = mg/k$ on the command line, but the numerical value depends on the values of $m$ and $g$ that we might also give in the command line. However, it is possible to specify `-S0 m*g/k` on the command line if we in the `init_prms` function first let `S0` be a string in the `elif` test and then, after the `for` loop, execute `S0 = eval(S0)`. At that point, `m` and `k` are read from the command line so that `eval` will work on `'m*g/k'`, or any other expression involving data from the command. Implement this idea.

A first test problem is to start from rest in the equilibrium position $S(0) = mg/k$ and give the plate a sudden upward change in position from $y = 0$ to $y = 1$. That is,

$$w(t) = \begin{cases} 0, & t \leq 0, \\ 1, & t > 0 \end{cases}$$

You should get oscillations around the displaced equilibrium position $Y = w - L - S_0 = -9 - 2g$. Name of program file: `box_spring2.py`. ◇

# Debugging

# D

## D.1 Using a Debugger

A debugger is a program that can help you to find out what is going on in a computer program. You can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until you track down abnormal behavior and find bugs.

Here we shall use the debugger to demonstrate the program flow of the `ball_table.py` code from Chapter 2.4.2. You are strongly encouraged to carry out the steps below on your computer to get a glimpse of what a debugger can do.

1. Go to the folder `src/basic` associated with Chapter 2.
2. Start IPython:

Terminal

```
Unix/DOS> ipython
```

3. Run the program `ball_table.py` with the debugger on (`-d`):

```
In [1]: run -d ball_table.py
```

Instead of running the program as usual, we now enter the debugger and get a prompt

```
ipdb>
```

After this prompt we can issue various debugger commands. The most important ones will be described as we go along.

4. Type `continue` or just `c` to go to the first statement in the file. Now you can see a printout of where we are in the program:

```
1---> 1 g = 9.81;  v0 = 5
      2 dt = 0.05
      3
```

Each program line is numbered and the arrow points to the next line to be executed. This is called the *current line*.

5. Type `step` or just `s` to execute the current line, which here initializes `g` and `v0`. Afterwards, we can print out their values by just writing the variable names at the `ipdb>` prompt:

```
ipdb>g
Out[1]: 9.8100000000000005
ipdb>v0
Out[1]: 5
```

We can explicitly demonstrate that `dt` is not yet initialized as a variable:

```
ipdb>dt
*** NameError: name 'dt' is not defined
```

6. Type `list` or just `l` to get a listing of the program lines around the current line:

```
ipdb>list
1     1 g = 9.81;  v0 = 5
----> 2 dt = 0.05
      3
      4 def y(t):
      5     return v0*t - 0.5*g*t**2
      6
      7 def table():
      8     data = []  # store [t, y] pairs in a nested list
      9     t = 0
     10     while y(t) >= 0:
     11         data.append([t, y(t)])
```

To see the lines between line number 11 and 30, type `list 11,30`. Writing `help list` results in a short description of the `list` command.

7. Let us set a *break point* at the line with the call `data = table()`. From the listing we see that this line has number 15. A break point means that the program execution will halt at this point to let us examine variables and perform step-wise execution with the `step` (`s`) command. Write

```
ipdb>break 15
```

to set the break point at line 15. To run the program up to this point, type `continue` or `c`. Alternatively, we could have issued some `step` commands to reach line 15.

8. Type repeated `step` (`s`) commands and see that we jump to the `table` function. Continue with step commands and observe that when we reach the `while` loop, we jump to the `y` function. After the `return` statement in the `y` function the debugger writes out the return value.

More step commands show how we jump between the `table` and `y` functions every time the `y(t)` expression appears in the loop (either in the append call or in the loop condition).

Inside the `y` or `table` function we may examine variables by just typing their names. One can, for instance, monitor how the `data` list develops. Inside the `y` function, `data` does not exist (since `data` is a local variable in the `table` not visible outside this function).

Stepwise execution with the `s` command is tedious. We may set a break point at line 11 when there are more elements in the `data` list, say when it has more than four elements:

```
ipdb>break 11, len(data)>4
```

Continue execution with the `continue` or `c` command and observe that the program stops at line 11 as soon as the condition on the length is fulfilled. Writing out the `data` list shows that it now has five elements.

9. The `next` or `n` command executes the current line, in the same way as the `step` or `s` command, but contrary to the latter, the `n` command does not enter functions being called. To demonstrate this point, type `n` a few times. You will experience that the statements involving `y(t)` calls are executed without stopping inside the `y` function. That is, `n` commands lead to stops inside the `table` function only. This is a quick way to examine how the `while` loop is executed.

10. Typing `c` continues execution until the next break point, but there are no more break points so the execution is continued until the end or until a Python error occurs. The latter action takes place in our program:

```
      8     data = []  # store [t, y] pairs in a nested list
      9     t = 0
---> 10     while y(t) >= 0:
     11         data.append([t, y(t)])
     12         t += dt

<type 'exceptions.TypeError'>: 'list' object is not callable
```

We can now check what `y` is by typing its name, and we quickly realize that `y` is a list, not a function anymore.

At this point, I hope you realize that a debugger is a very handy tool for monitoring the program flow, checking variables, and thereby understanding why errors occur, as we have demonstrated in the stepwise exploration above.

## D.2 How to Debug

Most programmers will claim that writing code consumes a small portion of the time it takes to develop a program – the major portion of

the work concerns testing the program and finding errors[1]. Newcomers to programming often panic when their program runs for the first time and aborts with a seemingly cryptic error message. How do you approach the art of debugging? This appendix summarizes some important working habits in this respect. Some of the tips are useful for problem solving in general, not only when writing and testing Python programs.

### D.2.1 A Recipe for Program Writing and Debugging

1. Make sure that you *understand the problem* the program is supposed to solve. We can make a general claim: If you do not understand the problem and the solution method, you will never be able to make a correct program[2]. It may be necessary to read a problem description or exercise many times and study relevant background material.
2. *Work out some examples* on input and output of the program. Such examples are important for controlling the understanding of the purpose of the program, and for verifying the implementation.
3. *Decide on a user interface*, i.e., how you want to get data into the program (command-line input, file input, questions and answers, etc.).
4. *Sketch rough algorithms* for various parts of the program. Some programmers prefer to do this on a piece of paper, others prefer to start directly in Python and write Python-like code with comments to sketch the program (this is easily developed into real Python code later).
5. *Look up information* on how to program different parts of the problem. Few programmers can write the whole program without consulting manuals, books, and the Internet. You need to know and understand the basic constructs in a language and some fundamental problem solving techniques, but technical details can be looked up.
   The more program examples you have studied (in this book, for instance), the easier it is to adapt ideas from an existing example to solve a new problem[3]. Remember that exercises in this book are often closely linked to examples in the text.

---

[1] "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." –Brian W. Kernighan, computer scientist, 1942-.

[2] This is not entirely true. Sometimes students with limited understanding of the problem are able to grab a similar program and guess at a few modifications – and get a program that works. But this technique is based on luck and not on understanding. The famous Norwegian computer scientist Kristen Nygaard (1926-2002) phrased it precisely: "Programming is understanding".

[3] "The secret to creativity is knowing how to hide your sources." –Albert Einstein, physicist, 1879-1955.

6. *Write the program.* Be extremely careful with what you write. In particular, compare all mathematical statements and algorithms with the original mathematical expressions on paper.
   In longer programs, do not wait until the program is complete before you start testing it – test parts while you write.
7. *Run the program.*
   If the program aborts with an error message from Python, these messages are fortunately quite precise and helpful. First, locate the line number where the error occurs and read the statement, then carefully read the error message. The most common errors (exceptions) are listed below.
   `SyntaxError`: Illegal Python code.

```
  File "somefile.py", line 5
    x = . 5
        ^
SyntaxError: invalid syntax
```

   Often the error is precisely indicated, as above, but sometimes you have to search for the error on the previous line.
   `NameError`: A name (variable, function, module) is not defined.

```
  File "somefile.py", line 20, in <module>
    table(10)
  File "somefile.py", line 16, in table
    value, next, error = L(x, n)
  File "somefile.py", line 8, in L
    exact_error = log(1+x) - value_of_sum
NameError: global name 'value_of_sum' is not defined
```

   Look at the last of the lines starting with `File` to see where in the program the error occurs. The most common reasons for a `NameError` are

   – a misspelled name,
   – a variable that is not initialized,
   – a function that you have forgotten to define,
   – a module that is not imported.

   `TypeError`: An object of wrong type is used in an operation.

```
  File "somefile.py", line 17, in table
    value, next, error = L(x, n)
  File "somefile.py", line 7, in L
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

   Print out objects and their types (here: `print x, type(x), n, type(n)`), and you will most likely get a surprise. The reason for a `TypeError` is often far away from the line where the `TypeError` occurs.
   `ValueError`: An object has an illegal value.

```
  File "somefile.py", line 8, in L
    y = sqrt(x)
ValueError: math domain error
```

Print out the value of objects that can be involved in the error (here: `print x`).

IndexError: An index in a list, tuple, string, or array is too large.

```
  File "somefile.py", line 21
    n = sys.argv[i+1]
IndexError: list index out of range
```

Print out the length of the list, and the index if it involves a variable (here: `print len(sys.argv), i`).

8. *Verify the implementation.* Assume now that we have a program that runs without error messages from Python. Before judging the results of the program, set precisely up a test case where you know the exact solution[4]. Insert `print` statements for all key results in the program so that you can easily compare calculations in the program with those done by hand.
   If your program produces wrong answers, start to *examine intermediate results.* Also remember that your hand calculations may be wrong!
9. If you need a lot of `print` statements in the program, you may *use a debugger* as explained in Appendix D.1.

Some may think that this list is very comprehensive. However, it just contains the items that you should always address when developing programs. Never forget that computer programming is a difficult task[5]!

### D.2.2 Application of the Recipe

Let us illustrate the points above in a specific programming problem.

*Problem.* Implement the Midpoint rule for numerical integration. The Midpoint rule for approximating an integral $\int_a^b f(x)dx$ reads

$$I = h\sum_{i=1}^{n} f(a + (i - \frac{1}{2})h), \quad h = \frac{b-a}{n}\,. \tag{D.1}$$

*Solution.* We just follow the individual steps in the recipe.

1. *Understand the problem.* In this problem we must understand how to program the formula (D.1). Observe that we do not need to understand how the formula is derived, because we do not apply the

[4] This is in general quite difficult. In complicated mathematical problems it is an art to construct good test problems and procedures for providing evidence that the program works.

[5] "Program writing is substantially more demanding than book writing." "Why is it so? I think the main reason is that a larger attention span is needed when working on a large computer program than when doing other intellectual tasks." –Donald Knuth [4, p. 18], computer scientist, 1938-.

derivation in the program[6]. What is important, is to notice that the formula is an *approximation* of an integral. If we try to integrate a function $f(x)$, we will probably not get an exact answer. Whether we have an approximation error or a programming error is always difficult to judge. We will meet this difficulty below.

2. *Work out examples.* As a test case we choose to integrate

$$f(x) = \sin^{-1}(x)\,. \tag{D.2}$$

between 0 and $\pi$. From a table of integrals we find that this integral equals

$$\left[x\sin^{-1}(x) + \sqrt{1-x^2}\,\right]_0^\pi\,. \tag{D.3}$$

The formula (D.1) gives an approximation to this integral, so the program will (most likely) print out a result different from (D.3). It would therefore be very helpful to construct a calculation where there are no approximation errors. Numerical integration rules usually integrate some polynomial of low order exactly. For the Midpoint rule it is obvious, if you understand the derivation of this rule, that a constant function will be integrated exactly. We therefore also introduce a test problem where we integrate $g(x) = 1$ from 0 to 10. The answer should be exactly 10.

*Input and output*: The input to the calculations is the function to integrate, the integration limits $a$ and $b$, and the $n$ parameter (number of intervals) in the formula (D.1). The output from the calculations is the approximation to the integral.

3. *User interface.* We decide to program the two functions $f(x)$ and $g(x)$ directly in the program. We also specify the corresponding integration limits $a$ and $b$ in the program, but we read a common $n$ for both integrals from the command line. Note that this is not a flexible user interface, but it suffices as a start for creating a working program. A much better user interface is to read $f$, $a$, $b$, and $n$ from the command line, which will be done later in a more complete solution to the present problem.

4. *Algorithm.* Like most mathematical programming problems, also this one has a generic part and an application part. The generic part is the formula (D.1), which is applicable to an arbitrary function $f(x)$. The implementation should reflect that we can specify any Python function `f(x)` and get it integrated. This principle calls for calculating (D.1) in a Python function where the input to the computation ($f$, $a$, $b$, $n$) are arguments. The function heading can look as `integrate(f, a, b, n)`, and the value of (D.1) is returned.

[6] You often need to understand the background for and the derivation of a mathematical formula in order to work out sensible test problems for verification. Sometimes this must be done by experts on the particular problem at hand.

The test part of the program consists of defining the test functions $f(x)$ and $g(x)$ and writing out the calculated approximations to the corresponding integrals.

A first rough sketch of the program can then be

```
def integrate(f, a, b, n):
    # compute integral, store in I
    return I

def f(x):
...

def g(x):
...

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
# calculate and print out the exact integral of f
```

The next step is to make a detailed implementation of the `integrate` function. Inside this function we need to compute the sum (D.1). In general, sums are computed by a `for` loop over the summation index, and inside the loop we calculate a term in the sum and add it to an accumulation variable. Here is the algorithm:

s = 0
for $i$ from 1 to $n$:
    $s = s + f(a + (i - \frac{1}{2})h)$
$I = sh$

5. *Look up information.* Our test function $f(x) = \sin^{-1}(x)$ must be evaluated in the program. How can we do this? We know that many common mathematical functions are offered by the `math` module. It is therefore natural to check if this module has an inverse sine function. The best place to look for Python modules is the Python Library Reference (see Chapter 2.4.3). We go to the index of this manual, find the "math" entry, and click on it. Alternatively, you can write `pydoc math` on the command line. Browsing the manual for the `math` module shows that there is an inverse sine function, with the name `asin`.

In this simple problem, where we use very basic constructs from the first three chapters of this book, there is hardly any need for looking at similar examples. Nevertheless, if you are uncertain about programming a mathematical sum, you may look at examples from, e.g., Chapter 2.2.4.

6. *Write the program.* Here is our first attempt to write the program. You can find the whole code in the file `appendix/integrate_v1.py`.

```
def integrate(f, a, b, n):
    s = 0
    for i in range(1, n):
        s += f(a + i*h)
    return s

def f(x):
return asin(x)

def g(x):
return 1

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10,  n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi,  n)
I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
print "Integral of f equals %g (exact value is %g)' % \
  (I, I_exact)
```

7. *Run the program.* We try a first execution from IPython

```
In [1]: run integrate_v1.py
```

Unfortunately, the program aborts with an error:

```
  File "integrate_v1.py", line 8
    return asin(x)
         ^
IndentationError: expected an indented block
```

We go to line 8 and look at that line and the surrounding code:

```
def f(x):
return asin(x)
```

Python expects that the return line is indented, because the function body must always be indented. By the way, we realize that there is a similar error in the `g(x)` function as well. We correct these errors:

```
def f(x):
    return asin(x)

def g(x):
    return 1
```

Running the program again makes Python respond with

```
  File "integrate_v1.py", line 24
    (I, I_exact)
               ^
SyntaxError: EOL while scanning single-quoted string
```

There is nothing wrong with line 24, but line 24 is a part of the statement starting on line 23:

```
print "Integral of f equals %g (exact value is %g)' % \
  (I, I_exact)
```

A `SyntaxError` implies that we have written illegal Python code. Inspecting line 23 reveals that the string to be printed starts with a

double quote, but ends with a single quote. We must be consistent and use the same enclosing quotes in a string. Correcting the statement,

```
print "Integral of f equals %g (exact value is %g)" % \
  (I, I_exact)
```

and rerunning the program yields the output

```
Traceback (most recent call last):
  File "integrate_v1.py", line 18, in <module>
    n = sys.argv[1]
NameError: name 'sys' is not defined
```

Obviously, we need to import `sys` before using it. We add `import sys` and run again:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 19, in <module>
    n = sys.argv[1]
IndexError: list index out of range
```

This is a very common error: We index the list `sys.argv` out of range because we have not provided enough command-line arguments. Let us use $n = 10$ in the test and provide that number on the command line:

```
In [5]: run integrate_v1.py 10
```

We still have problems:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10,  n)
  File "integrate_v1.py", line 7, in integrate
    for i in range(1, n):
TypeError: range() integer end argument expected, got str.
```

It is the final `File` line that counts (the previous ones describe the nested functions calls up to the point where the error occured). The error message for line 7 is very precise: The end argument to `range`, `n`, should be an integer, but it is a string. We need to convert the string `sys.argv[1]` to `int` before sending it to the `integrate` function:

```
n = int(sys.argv[1])
```

After a new edit-and-run cycle we have other error messages waiting:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10,  n)
  File "integrate_v1.py", line 8, in integrate
    s += f(a + i*h)
NameError: global name 'h' is not defined
```

The `h` variable is used without being assigned a value. From the formula (D.1) we see that $h = (b - a)/n$, so we insert this assignment at the top of the `integrate` function:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    ...
```

A new run results in a new error:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 23, in <module>
    I = integrate(f, 0, pi,  n)
NameError: name 'pi' is not defined
```

Looking carefully at all output, we see that the program managed to call the `integrate` function with `g` as input and write out the integral. However, in the call to `integrate` with `f` as argument, we get a `NameError`, saying that `pi` is undefined. When we wrote the program we took it for granted that `pi` was $\pi$, but we need to import `pi` from `math` to get this variable defined, before we call `integrate`:

```
from math import pi
I = integrate(f, 0, pi,  n)
```

The output of a new run is now

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
NameError: global name 'asin' is not defined
```

A similar error occured: `asin` is not defined as a function, and we need to import it from `math`. We can either do a

```
from math import pi, asin
```

or just do the rough

```
from math import *
```

to avoid any further errors with undefined names from the `math` module (we will get one for the `sqrt` function later, so we simply use the last "import all" kind of statement).

There are still more errors:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
ValueError: math domain error
```

Now the error concerns a wrong `x` value in the `f` function. Let us print out `x`:

```
def f(x):
    print x
    return asin(x)
```

The output becomes

```
Integral of g equals 9
0.314159265359
0.628318530718
0.942477796077
1.25663706144
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I = integrate(f, 0, pi,  n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 14, in f
    return asin(x)
ValueError: math domain error
```

We see that all the `asin(x)` computations are successful up to and including $x = 0.942477796077$, but for $x = 1.25663706144$ we get an error. A "math domain error" may point to a wrong $x$ value for $\sin^{-1}(x)$ (recall that the domain of a function specifies the legal $x$ values for that function).

To proceed, we need to think about the mathematics of our problem: Since $\sin(x)$ is always between $-1$ and 1, the inverse sine function cannot take $x$ values outside the interval $[-1, 1]$. The problem is that we try to integrate $\sin^{-1}(x)$ from 0 to $\pi$, but only integration limits within $[-1, 1]$ make sense (unless we allow for complex-valued trigonometric functions). Our test problem is hence wrong from a mathematical point of view. We need to adjust the limits, say 0 to 1 instead of 0 to $\pi$. The corresponding program modification reads

```
I = integrate(f, 0, 1,  n)
```

We run again and get

```
Integral of g equals 9
0
0
0
0
0
0
0
0
0
Traceback (most recent call last):
  File "integrate_v1.py", line 26, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

It is easy to go directly to the `ValueError` now, but one should always examine the output from top to bottom. If there is strange output before Python reports an error, there may be an error indicated by our `print` statements which causes Python to abort the program. This is not the case in the present example, but it is a good habit to start at the top of the output anyway. We see that all our `print x` statements inside the `f` function say that `x` is zero. This must be wrong – the idea

of the integration rule is to pick $n$ different points in the integration interval $[0, 1]$.

Our `f(x)` function is called from the `integrate` function. The argument to `f`, `a + i*h`, is seemingly always 0. Why? We print out the argument and the values of the variables that make up the argument:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    s = 0
    for i in range(1, n):
        print a, i, h, a+i*h
        s += f(a + i*h)
    return s
```

Running the program shows that `h` is zero and therefore `a+i*h` is zero.

Why is `h` zero? We need a new `print` statement in the computation of `h`:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    print b, a, n, h
    ...
```

The output shows that `a`, `b`, and `n` are correct. Now we have encountered an error that we often discuss in this book: integer division (see Chapter 1.3.1). The formula $(1-0)/10 = 1/10$ is zero according to integer division. The reason is that `a` and `b` are specified as `0` and `1` in the call to `integrate`, and `0` and `1` imply `int` objects. Then `b-a` becomes an `int`, and `n` is an int, causing an `int`/`int` division. We must ensure that `b-a` is `float` to get the right mathematical division in the computation of `h`:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    ...
```

Thinking that the problem with wrong $x$ values in the inverse sine function is resolved, we may remove all the `print` statements in the program, and run again.

The output now reads

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

That is, we are back to the `ValueError` we have seen before. The reason is that `asin(pi)` does not make sense, and the argument to `sqrt` is negative. The error is simply that we forgot to adjust the upper integration limit in the computation of the exact result. This is another very common error. The correct line is

```
I_exact = 1*asin(1) - sqrt(1 - 1**2) - 1
```

We could avoided the error by introducing variables for the integration limits, and a function for $\int f(x)dx$ would make the code cleaner:

```
a = 0; b = 1
def int_f_exact(x):
    return x*asin(x) - sqrt(1 - x**2)
I_exact = int_f_exact(b) - int_f_exact(a)
```

Although this is more work than what we initially aimed at, it usually saves time in the debugging phase to do things this proper way.

Eventually, the program seems to work! The output is just the result of our two `print` statements:

```
Integral of g equals 9
Integral of f equals 5.0073 (exact value is 0.570796)
```

8. *Verify the results.* Now it is time to check if the numerical results are correct. We start with the simple integral of 1 from 0 to 10: The answer should be 10, not 9. Recall that for this particular choice of integration function, there is no approximation error involved (but there could be a small round-off error). Hence, there must be a programming error.

To proceed, we need to calculate some intermediate mathematical results by hand and compare these with the corresponding statements in the program. We choose a very simple test problem with $n = 2$ and $h = (10 - 0)/2 = 5$. The formula (D.1) becomes

$$I = 5 \cdot (1 + 1) = 10 \,.$$

Running the program with $n = 2$ gives

```
Integral of g equals 1
```

We insert some `print` statements inside the `integrate` function:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    s = 0
    for i in range(1, n):
        print 'i=%d, a+i*h=%g' % (i, a+i*h)
        s += f(a + i*h)
    return s
```

Here is the output:

```
i=1, a+i*h=5
Integral of g equals 1
i=1, a+i*h=0.5
Integral of f equals 0.523599 (exact value is 0.570796)
```

There was only one pass in the `i` loop in `integrate`. According to the formula, there should be $n$ passes, i.e., two in this test case. The limits of `i` must be wrong. The limits are produced by the call `range(1,n)`. We recall that such a call results in integers going from 1 up to `n`, but

*not* including `n`. We need to include `n` as value of `i`, so the right call to `range` is `range(1,n+1)`.

We make this correction and rerun the program. The output is now

```
i=1, a+i*h=5
i=2, a+i*h=10
Integral of g equals 2
i=1, a+i*h=0.5
i=2, a+i*h=1
Integral of f equals 2.0944 (exact value is 0.570796)
```

The integral of 1 is still not correct. We need more intermediate results!

In our quick hand calculation we knew that $g(x) = 1$ so all the $f(a + (i - \frac{1}{2})h)$ evaluations were rapidly replaced by ones. Let us now compute all the $x$ coordinates $a+(i-\frac{1}{2})h$ that are used in the formula:

$$i = 1: \ a + (i - \frac{1}{2})h = 2.5, \quad i = 2: \ a + (i - \frac{1}{2})h = 7.5\,.$$

Looking at the output from the program, we see that the argument to `g` has a different value – and fortunately we realize that the formula we have coded is wrong. It should be `a+(i-0.5)*h`.

We correct this error and run the program:

```
i=1, a+(i-0.5)*h=2.5
i=2, a+(i-0.5)*h=7.5
Integral of g equals 2
...
```

Still the integral is wrong[7].

Now we read the code more carefully and compare expressions with those in the mathematical formula. We should, of course, have done this already when writing the program, but it is easy to get excited when writing code and hurry for the end. This ongoing story of debugging probably shows that reading the code carefully can save much debugging time[8]. We clearly add up all the $f$ evaluations correctly, but then this sum must be multiplied by $h$, and we forgot that in the code. The `return` statement in `integrate` must therefore be modified to

```
    return s*h
```

Eventually, the output is

```
Integral of g equals 10
Integral of f equals 0.568484 (exact value is 0.570796)
```

and we have managed to integrate a constant function in our program! Even the second integral looks promising!

To judge the result of integrating the inverse sine function, we need to run several increasing $n$ values and see that the approximation gets

[7] At this point you may give up programming, but the more skills you pick up in debugging, the more fun it is to hunt for errors! Debugging is like reading an exciting criminal novel: the detective follows different ideas and tracks, but never gives up before the culprit is caught.

[8] Actually, being extremely careful with what you write, and comparing all formulas with the mathematics, may be the best way to get more spare time when taking a programming course!

better. For $n = 2, 10, 100, 1000$ we get 0.550371, 0.568484, 0.570714, 0.570794, to be compared to the exact[9] value 0.570796. The decreasing error provides evidence for a correct program, but it is not a strong proof. We should try out more functions. In particular, linear functions are integrated exactly by the Midpoint rule. We can also measure the speed of the decrease of the error and check that the speed is consistent with the properties of the Midpoint rule, but this is a mathematically more advanced topic.

The very important lesson learned from these debugging sessions is that you should start with a simple test problem where all formulas can be computed by hand. If you start out with $n = 100$ and try to integrate the inverse sine function, you will have a much harder job with tracking down all the errors.

9. *Use a debugger.* Another lesson learned from these sessions is that we needed many `print` statements to see intermediate results. It is an open question if it would be more efficient to run a debugger and stop the code at relevant lines. In an edit-and-run cycle of the type we met here, we frequently need to examine many numerical results, correct something, and look at all the intermediate results again. Plain `print` statements are often better suited for this massive output than the pure manual operation of a debugger, unless one writes a program to automate the interaction with the debugger.

The correct code for the implementation of the Midpoint rule is found in `integrate_v2.py`. Some readers might be frightened by all the energy it took to debug this code, but this is just the nature of programming. The experience of developing programs that finally work is very awarding[10].

*Refining the User Interface.* We briefly mentioned that the chosen user interface, where the user can only specify $n$, is not particularly user friendly. We should allow $f$, $a$, $b$, and $n$ to be specified on the command line. Since $f$ is a function and the command line can only provide strings to the program, we may use the `StringFunction` object from `scitools.std` to convert a string expression for the function to be integrated to an ordinary Python function (see Chapter 3.1.4). The other parameters should be easy to retrieve from the command line if Chapter 3.2 is understood. As suggested in Chapter 3.3, we enclose the input statements in a `try-except` block, here with a specific exception type `IndexError` (because an index in `sys.argv` out of bounds is the only type of error we expect to handle):

[9] This is not the mathematically exact value, because it involves computations of $\sin^{-1}(x)$, which is only approximately calculated by the `asin` function in the `math` module. However, the approximation error is very small ($\sim 10^{-16}$).

[10] "People only become computer programmers if they're obsessive about details, crave power over machines, and can bear to be told day after day exactly how stupid they are." –Gregory J. E. Rawlins, computer scientist. Quote from the book "Slaves of the Machine: The Quickening of Computer Technology", MIT Press, 1997.

```
try:
    f_formula = sys.argv[1]
    a = eval(sys.argv[2])
    b = eval(sys.argv[3])
    n = int(sys.argv[4])
except IndexError:
    print 'Usage: %s f-formula a b n' % sys.argv[0]
    sys.exit(1)
```

Note that the use of `eval` allows us to specify `a` and `b` as `pi` or `exp(5)` or another mathematical expression.

With the input above we can perform the general task of the program:

```
from scitools.std import StringFunction
f = StringFunction(f_formula)
I = integrate(f, a, b, n)
print I
```

Instead of having these test statements as a main program we follow the good habits of Chapter 3.5 and make a module with (i) the `integrate` function, (ii) a `verify` function for testing the `integrate` function's ability to exactly integrate linear functions, and (iii) a `main` function for reading data from the command line and calling `integrate` for the user's problem at hand. Any module should also have a test block, and doc strings for the module itself and all functions.

The `verify` function performs a loop over some specified `n` values and checks that the Midpoint rule integrates a linear function exactly[11]. In the test block we can either run the `verify` function or the `main` function.

The final solution to the problem of implementing the Midpoint rule for numerical integration is now the following complete module file `integrate.py`:

```
"""Module for integrating functions by the Midpoint rule."""
from math import *
import sys

def integrate(f, a, b, n):
    """Return the integral of f from a to b with n intervals."""
    h = float(b-a)/n
    s = 0
    for i in range(1, n+1):
        s += f(a + (i-0.5)*h)
    return s*h

def verify():
    """Check that linear functions are integrated exactly."""

    def g(x):
        return p*x + q   # general linear function

    def int_g_exact(x):  # integral of g(x)
        return 0.5*p*x**2 + q*x
```

[11] We must be prepared for round-off errors, so "exactly" means errors less than (say) $10^{-14}$.

```
    a = -1.2; b = 2.8    # "arbitrary" integration limits
    p = -2;   q = 10
    passed = True        # True if all tests below are passed
    for n in 1, 10, 100:
        I = integrate(g, a, b, n)
        I_exact = int_g_exact(b) - int_g_exact(a)
        error = abs(I_exact - I)
        if error > 1E-14:
            print 'Error=%g for n=%d' % (error, n)
            passed = False
    if passed: print 'All tests are passed.'


def main():
    """
    Read f-formula, a, b, n from the command line.
    Print the result of integrate(f, a, b, n).
    """
    try:
        f_formula = sys.argv[1]
        a = eval(sys.argv[2])
        b = eval(sys.argv[3])
        n = int(sys.argv[4])
    except IndexError:
        print 'Usage: %s f-formula a b n' % sys.argv[0]
        sys.exit(1)

    from scitools.std import StringFunction
    f = StringFunction(f_formula)
    I = integrate(f, a, b, n)
    print I

if __name__ == '__main__':
    if sys.argv[1] == 'verify':
        verify()
    else:
        # compute the integral specified on the command line:
        main()
```

Here is a short demo computing $\int_0^{2\pi}(\cos(x) + \sin(x))dx$:

Terminal

```
integrate.py 'cos(x)+sin(x)' 0 2*pi 10
-3.48786849801e-16
```

# Technical Topics

# E

## E.1 Different Ways of Running Python Programs

Python programs are compiled and interpreted by another program called `python`. To run a Python program, you need to tell the operating system that your program is to be interpreted by the `python` program. This section explains various ways of doing this.

### E.1.1 Executing Python Programs in IPython

The simplest and most flexible way of executing a Python program is to run it inside IPython. See Chapter 1.5.3 for a quick introduction to IPython. You start IPython either by the command `ipython` in a terminal window, or by double-clicking the IPython program icon (on Windows). Then, inside IPython, you can run a program `prog.py` by

```
In [1]: run prog.py arg1 arg2
```

where `arg1` and `arg2` are command-line arguments.

This method of running Python programs works the same way on all platforms. One additional advantage of running programs under IPython is that you can automatically enter the Python debugger if an exception is raised (see Appendix D.1. Although we advocate running Python programs under IPython in this book, you can also run them directly under specific operating systems. This is explained next for Unix, Windows, and Mac OS X.

### E.1.2 Executing Python Programs on Unix

There are two ways of executing a Python program `prog.py` on Unix. The first explicitly tells which Python interpreter to use:

Terminal
```
Unix> python prog.py arg1 arg2
```

Here, `arg1` and `arg2` are command-line arguments.

There may be many Python interpreters on your computer system, usually corresponding to different versions of Python or different sets of additional packages and modules. The Python interpreter (`python`) used in the command above is the first program with the name `python` appearing in the folders listed in your `PATH` environment variable. A specific `python` interpreter, say in `/home/hpl/local/bin`, can easily be used to run a program `prog.py` in the current working folder by specifying the interpreter's complete filepath:

Terminal
```
Unix> /home/hpl/bin/python prog.py arg1 arg2
```

The other way of executing Python programs on Unix consists of just writing the name of the file:

Terminal
```
Unix> ./prog.py arg1 arg2
```

The leading `./` is needed to tell that the program is located in the current folder. You can also just write

Terminal
```
Unix> prog.py arg1 arg2
```

but then you need to have the dot[1] in the `PATH` variable, and this is not recommended of security reasons.

In the two latter commands there is no information on which Python interpreter to use. This information must be provided in the first line of the program, normally as

```
#!/usr/bin/env python
```

This looks like a comment line, and behaves indeed as a comment line when we run the program as `python prog.py`. However, when we run the program as `./prog.py`, the first line beginning with `#!` tells the operating system to use the program specified in the rest of the first line to interpret the program. In this example, we use the first `python` program encountered in the folders in your `PATH` variable. Alternatively, a specific `python` program can be specified as

[1] The dot acts as the name of the current folder (usually known as the current working directory). A double dot is the name of the parent folder.

```
#!/home/hpl/special/tricks/python
```

It is a good habit to always include such a first line (also called shebang line) in all Python programs and modules, but we have not done that in this book.

### E.1.3 Executing Python Programs on Windows

In a DOS window you can always run a Python program by

Terminal
```
DOS> python prog.py arg1 arg2
```

if `prog.py` is the name of the program, and `arg1` and `arg2` are command-line arguments. The extension `.py` can be dropped:

Terminal
```
DOS> python prog arg1 arg2
```

If there are several Python installations on your system, a particular installation can be specified:

Terminal
```
DOS> E:\hpl\myprogs\Python2.5.3\python prog arg1 arg2
```

Files with a certain extension can on Windows be associated with a file type, and a file type can be associated with a particular program to handle the file. For example, it is natural to associate the extension `.py` with Python programs. The corresponding program needed to interpret `.py` files is then `python.exe`. When we write just the name of the Python program file, as in

Terminal
```
DOS> prog arg1 arg2
```

the file is always interpreted by the specified `python.exe` program. The details of getting `.py` files to be interpreted by `python.exe` go as follows:

Terminal
```
DOS> assoc .py=PyProg
DOS> ftype PyProg=python.exe "%1" %*
```

Depending on your Python installation, such file extension bindings may already be done. You can check this with

Terminal
```
DOS> assoc | find "py"
```

To see the programs associated with a file type, write `ftype name` where `name` is the name of the file type as specified by the `assoc` command. Writing `help ftype` and `help assoc` prints out more information about these commands along with examples.

One can also run Python programs by writing just the basename of the program file, i.e., `prog.py` instead of `prog.py`, if the file extension is registered in the `PATHEXT` environment variable.

*Double-Clicking Python Files.* The usual way of running programs on Windows is to double click on the file icon. This does not work well with Python programs without a graphical user interface. When you double click on the icon for a file `prog.py`, a DOS window is opened, `prog.py` is interpreted by some `python.exe` program, and when the program terminates, the DOS window is closed. There is usually too little time for the user to observe the output in this short-lived DOS window.

One can always insert a final statement that pauses the program by waiting for input from the user:

```
raw_input('Type CR:')
```

or

```
sys.stdout.write('Type CR:'); sys.stdin.readline()
```

The program will "hang" until the user presses the Return key. During this pause the DOS window is visible and you can watch the output from previous statements in the program.

The downside of including a final input statement is that you must always hit Return before the program terminates. This is inconvenient if the program is moved to a Unix-type machine. One possibility is to let this final input statement be active only when the program is run on Windows:

```
if sys.platform[:3] == 'win':
    raw_input('Type CR:')
```

Python programs that have a graphical user interface can be double-clicked in the usual way if the file extension is `.pyw`.

*Gnuplot Plots on Windows.* Programs that call `plot` to visualize a graph with the aid of Gnuplot suffer from the same problem as described above: the plot window disappears quickly. Again, the recipe is to insert a `raw_input` call at the end of the program.

### E.1.4 Executing Python Programs on Macintosh

Since a variant of Unix is used as core in the Mac OS X operating system, you can always launch a Unix terminal and use the techniques from Appendix E.1.2 to run Python programs.

### E.1.5 Making a Complete Stand-Alone Executable

Python programs need a Python interpreter and usually a set of modules to be installed on the computer system. Sometimes this is inconvenient, for instance when you want to give your program to somebody who does not necessarily have Python or the required set of modules installed.

Fortunately, there are tools that can create a stand-alone executable program out of a Python program. This stand-alone executable can be run on every computer that has the same type of operating system and the same chip type. Such a stand-alone executable is a bundling of the Python interpreter and the required modules, along with your program, in a single file. Details of producing this single file are given in the book [9].

## E.2 Integer and Float Division

Many languages, including C, C++, Fortran, and classical Python, interpret the division operator in two ways:

1. *Integer division:* If both operands `a` and `b` are integers, the result `a/b` is the *floor* of the mathematical result `a/b`. This yields the largest integer that `b` can be multiplied with such that the product is less than or equal to `a`. Or phrased simpler: The result of `a/b` is an integer which is "rounded down". As an example, 5/2 becomes 2.
2. *Float division:* If one of the operands is a floating-point number or a complex number, `a/b` returns the mathematical result of the division.

Accidental integer division in places where mathematical division is needed, constitutes a very common source of errors in numerical programs.

It is often argued that in a statically typed language, where each variable is declared with a fixed type, the programmer always knows the type of the operands involved in a division expression. Therefore the programmer can determine whether an expression has the right form or not (the programmer can still oversee such errors). In a dynamically typed language, such as Python, variables can hold objects of any type. If `a` or `b` is provided by the user of the program, one can never know if both types end up as integer and `a/b` will imply integer division.

The only safe solution is to have two different operands for integer division and mathematical division. Python is currently moving in this direction. By default, `a/b` still has its original double meaning, depending on the types of operands. A new operator `//` is introduced for explicitly employing integer division. To force `a/b` to mean standard mathematical float division, one can write

```
from __future__ import division
```

This import statement must be present in every module file or script where the `/` operator always shall imply float division. Alternatively, one can run a Python program `someprogram.py` from the command line with the argument `-Qnew` to the Python interpreter:

```
Unix/DOS> python -Qnew someprogram.py
```

The future Python 3.0 is suggested to abandon integer division interpretation of `a/b`, i.e., `a/b` will always mean the relevant float division, depending on the operands (float division for `int` and `float` operands, and complex division if one of the operands is a `complex`).

Running a Python program with the `-Qwarnall` argument, say

```
Unix/DOS> python -Qwarnall someprogram.py
```

will print out a warning every time an integer division expression is encountered.

There are currently alternative ways out of the integer division problem:

1. If the operands involve an integer with fixed value, such as in `a/2`, the integer can be written as a floating-point number, as in `a/2.0` or `a/2.`, to enforce mathematical division regardless of whether `a` is integer, float, or complex.
2. If both operands are variables, as in `a/b`, the only safe way out of the problem is to write `1.0*a/b`. Note that `float(a)/b` or `a/float(b)` will work correctly from a mathematical viewpoint if `a` and `b` are of integer or floating-point type, but not if the argument to `float` is complex.

## E.3 Visualizing a Program with Lumpy

Lumpy is a nice tool for graphically displaying the relations between the variables in a program. Consider the following program (inspired by Chapter 2.1.9), where we extract a sublist and modify the original list:

```
l0 = [1, 4, 3]
l1 = l0
l2 = l1[:-1]
l1[0] = 100
```

The point is that the change in l1 is reflected in l0, but not in l2, because sublists are created by taking a copy of the original list, while l1 and l0 refer to the same object. Lumpy can visually display the variables and how they relate, and thereby making it obvious that l0 and l1 refer to the same object and that l2 is a different object. To use Lumpy, some extra statements must be inserted in the program:

```
from scitools.Lumpy import Lumpy
lumpy = Lumpy()
lumpy.make_reference()
l0 = [1, 4, 3]
l1 = l0
l2 = l1[:-1]
l1[0] = 100
lumpy.object_diagram()
```

By running this program a graphical window is shown on the screen with the variables in the program, see Figure E.1a. The variables have lines to the object they point to, and inside the objects we can see the contents, i.e., the list elements in this case.

We can add some lines to the program above and make a new, additional drawing:

```
lumpy = Lumpy()
lumpy.make_reference()
n1 = 21.5
n2 = 21
l3 = [l1, l2, [n1, n2]]
s1 = 'some string'
lumpy.object_diagram()
```

Figure E.1b shows the second object diagram with the additional variables.

We recommend to actively use Lumpy to make graphical illustrations of programs, especially if you search for an error and you are not 100% sure of how all variables related to each other.

## E.4 Doing Operating System Tasks in Python

Python has extensive support for operating system tasks, such as file and folder management. The great advantage of doing operating system tasks in Python and not directly in the operating system is that the Python code works uniformly on Unix/Linux, Windows, and Mac (there are exceptions, but they are few). Below we list some useful operations that can be done inside a Python program or in an interactive session.

**Make a folder:**

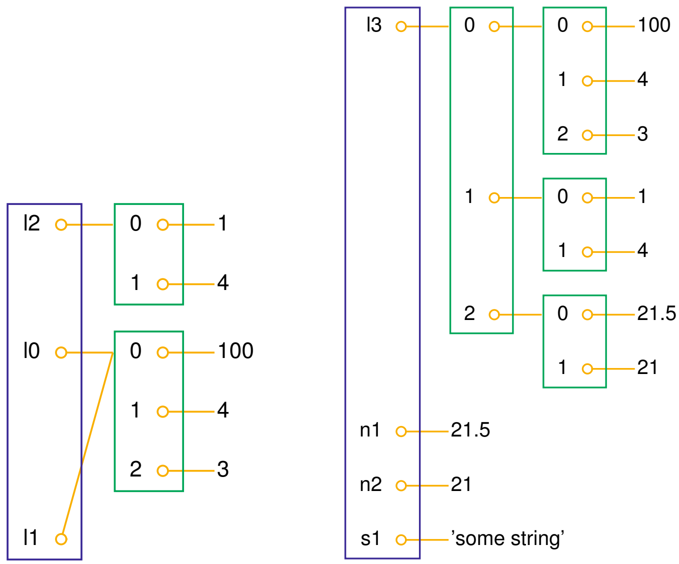



**Fig. E.1** Output from Lumpy: (a) program with three lists; (b) extended program with another list, two floats, and a string.

```
import os
os.mkdir(foldername)
```

Recall that Python applies the term directory instead of folder. Ordinary files are created by the `open` and `close` functions in Python.

**Make intermediate folders:** Suppose you want to make a subfolder under your home folder:

`$HOME/python/project1/temp`

but the intermediate folders `python` and `project1` do not exist. This requires each new folder to be made separately by `os.mkdir`, or you can make all folders at once with `os.makedirs`:

```
foldername = os.path.join(os.environ['HOME'], 'python',
                          'project1', 'temp')
os.makedirs(foldername)
```

With `os.environ[var]` we can get the value of any environment variable `var` as a string.

**Move to a folder:**

```
origfolder = os.getcwd()  # get name of current folder
os.chdir(foldername)      # move ("change directory")
...
os.chdir(origfolder)      # move back
```

**Rename a file or folder:**

```
os.rename(oldname, newname)
```

**List files** (using Unix shell wildcard notation):

```
import glob
filelist1 = glob.glob('*.py')
filelist2 = glob.glob('*[1-4]*.dat')
```

**List all files and folders in a folder:**

```
filelist1 = os.listdir(foldername)
filelist1 = os.listdir(os.curdir)  # current folder (directory)
```

**Check if a file or folder exists:**

```
if os.path.isfile(filename):
    f = open(filename)
    ...

if os.path.isdir(foldername):
    filelist = os.listdir(foldername)
    ...
```

**Remove files:**

```
import glob
filelist = glob.glob('tmp_*.eps')
for filename in filelist:
    os.remove(filename)
```

**Remove a folder and all its subfolders:**

```
import shutil
shutil.rmtree(foldername)
```

It goes without saying that this command may be dangerous!

**Copy a file to another file or folder:**

```
shutil.copy(sourcefile, destination)
```

**Copy a folder and all its subfolders:**

```
shutil.copytree(sourcefolder, destination)
```

**Run any operating system command:**

```
cmd = 'c2f.py 21'   # command to be run
failure = os.system(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd
```

```
    sys.exit(1)

# record output from the command:
from subprocess import Popen, PIPE
p = Popen(cmd, shell=True, stdout=PIPE)
output, errors = p.communicate()
# output contains text sent to standard output
# errors contains text sent to standard error

# process output:
for line in output.splitlines():
    # process line

# simpler recording of output on Linux/Unix:
import commands
failure, output = commands.getstatusoutput(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd, output
    sys.exit(1)
```

The constructions above are mainly used for running stand-alone programs. Any file or folder listing or manipulation should be done by the functionality in `os` or other modules.

**Split file or folder name:**

```
>>> fname = os.path.join(os.environ['HOME'], 'data', 'file1.dat')
>>> foldername, basename = os.path.split(fname)
>>> foldername
'/home/hpl/data'
>>> basename
'file1.dat'
>>> outfile = basename[:-4] + '.out'
>>> outfile
'file1.out'
```

## E.5 Variable Number of Function Arguments

Arguments to Python functions are of four types:

1. positional arguments, where each argument has a name,
2. keyword arguments, where each argument has a name and a default value,
3. a variable number of positional arguments, where each argument has no name, but just a location in a list,
4. a variable number of keyword arguments, where each argument is a (name, default value) pair in a dictionary.

The corresponding general function definition can be sketched as

```
def f(pos1, pos2, key1=val1, key2=val2, *args, **kwargs):
```

Here, `pos1` and `pos2` are positional arguments, `key1` and `key2` are keyword arguments, `args` is a tuple holding a variable number of positional

arguments, and `kwargs` is a dictionary holding a variable number of keyword arguments. This appendix describes how to program with the `args` and `kwargs` variables and why these are handy in many situations.

### E.5.1 Variable Number of Positional Arguments

Let us start by making a function that takes an arbitrary number of arguments and computes their sum:

```
>>> def add(*args):
...     print 'args:', args
...     return sum(args)
...
>>> add(1)
args: (1,)
1
>>> add(1,5,10)
args: (1, 5, 10)
16
```

We observe that `args` is a tuple and that all the arguments we provide in a call to `add` are stored in `args`.

Combination of ordinary positional arguments and a variable number of arguments is allowed, but the `*args` argument must appear after the ordinary positional arguments, e.g.,

```
def f(pos1, pos2, pos3, *args):
```

In each call to `f` we must provide at least three arguments. If more arguments are supplied in the call, these are collected in the `args` tuple inside the `f` function.

*Example.* Chapter 7.1.1 describes functions with parameters, e.g., $y(t; v_0) = v_0 t - \frac{1}{2}gt^2$, or the more general case $f(x; p_1, \ldots, p_n)$. The Python implementation of such functions can take both the independent variable and the parameters as arguments: `y(t, v0)` and `f(x, p1, p2, ...,pn)`. Suppose that we have a general library routine that operates on functions of one variable. Relevant operations can be numerical differentiation, integration, or root finding. A simple example is a numerical differentiation function

```
def diff(f, x, h):
    return (f(x+h) - f(x))/h
```

This `diff` function cannot be used with functions `f` that take more than one argument, e.g., passing an `y(t, v0)` function as `f` leads to the exception

```
TypeError: y() takes exactly 2 arguments (1 given)
```

Chapter 7.1.1 provides a solution to this problem where `y` becomes a class instance. Here we can describe an alternative solution that allows our `y(t, v0)` function to be used as is.

The idea is that we pass additional arguments for the parameters in the `f` function *through* the `diff` function. That is, we view the `f` function as `f(x, *f_prms)`. Our `diff` routine can then be written as

```
def diff(f, x, h, *f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, *f_prms) - f(x, *f_prms))/h
```

Before explaining this function in detail, we "prove" that it works in an example:

```
def y(t, v0):
    g = 9.81; return v0*t - 0.5*g*t**2

dydt = diff(y, 0.1, 1E-9, 3)  # t=0.1, h=1E-9, v0=3
```

The output from the call to `diff` becomes

```
x: 0.1 h: 1e-09 f_prms: (3,)
```

The point is that the `v0` parameter, which we want to pass on to our `y` function, is now stored in `f_prms`. Inside the `diff` function, calling

```
f(x, *f_prms)
```

is the same as if we had written

```
f(x, f_prms[0], f_prms[1], ...)
```

That is, `*f_prms` in a call takes all the values in the tuple `*f_prms` and places them after each other as positional arguments. In the present example with the `y` function, `f(x, *f_prms)` implies `f(x, f_prms[0])`, which for the current set of argument values in our example becomes a call `y(0.1, 3)`.

For a function with many parameters,

```
def G(x, t, A, a, w):
    return A*exp(-a*t)*sin(w*x)
```

the output from

```
dGdx = diff(G, 0.5, 1E-9, 0, 1, 0.6, 100)
```

becomes

```
x: 0.5 h: 1e-09 f_prms: (0, 1, 1.5, 100)
```

We pass here the arguments `t`, `A`, `a`, and `w`, in that sequence, as the last four arguments to `diff`, and all the values are stored in the `f_prms` tuple.

The `diff` function also works for a plain function `f` with one argument:

```
from math import sin
mycos = diff(sin, 0, 1E-9)
```

In this case, `*f_prms` becomes an empty tuple, and a call like `f(x, *f_prms)` is just `f(x)`.

The use of a variable set of arguments for sending problem-specific parameters "through" a general library function, as we have demonstrated here with the `diff` function, is perhaps the most frequent use of `*args`-type arguments.

### E.5.2 Variable Number of Keyword Arguments

A simple test function

```
>>> def test(**kwargs):
...     print kwargs
```

exemplifies that `kwargs` is a dictionary inside the `test` function, and that we can pass any set of keyword arguments to `test`, e.g.,

```
>>> test(a=1, q=9, method='Newton')
{'a': 1, 'q': 9, 'method': 'Newton'}
```

We can combine an arbitrary set of positional and keyword arguments, provided all the keyword arguments appear at the end of the call:

```
>>> def test(*args, **kwargs):
...     print args, kwargs
...
>>> test(1,3,5,4,a=1,b=2)
(1, 3, 5, 4) {'a': 1, 'b': 2}
```

From the output we understand that all the arguments in the call where we provide a name and a value are treated as keyword arguments and hence placed in `kwargs`, while all the remaining arguments are positional and placed in `args`.

*Example.* We may extend the example in Appendix E.5.1 to make use of a variable number of keyword arguments instead of a variable number of positional arguments. Suppose all functions with parameters in addition to an independent variable take the parameters as keyword arguments. For example,

```
def y(t, v0=1):
    g = 9.81; return v0*t - 0.5*g*t**2
```

In the `diff` function we transfer the parameters in the `f` function as a set of keyword arguments `**f_prms`:

```
def diff(f, x, h=1E-10, **f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, **f_prms) - f(x, **f_prms))/h
```

In general, the `**f_prms` argument in a call

```
f(x, **f_prms)
```

implies that all the key-value pairs in `**f_prms` are provided as keyword arguments:

```
f(x, key1=f_prms[key1], key2=f_prms[key2], ...)
```

In our special case with the `y` function and the call

```
dydt = diff(y, 0.1, h=1E-9, v0=3)
```

`f(x, **f_prms)` becomes `y(0.1, v0=3)`. The output from `diff` is now

```
x: 0.1 h: 1e-09 f_prms: {'v0': 3}
```

showing explicitly that our `v0=3` in the call to `diff` is placed in the `f_prms` dictionary.

The `G` function from Appendix E.5.1 can also have its parameters as keyword arguments:

```
def G(x, t=0, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

We can now make the call

```
dGdx = diff(G, 0.5, h=1E-9, t=0, A=1, w=100, a=1.5)
```

and view the output from `diff`,

```
x: 0.5 h: 1e-09 f_prms: {'A': 1, 'a': 1.5, 't': 0, 'w': 100}
```

to see that all the parameters get stored in `f_prms`. The `h` parameter can be placed anywhere in the collection of keyword arguments, e.g.,

```
dGdx = diff(G, 0.5, t=0, A=1, w=100, a=1.5, h=1E-9)
```

We can allow the `f` function of one variable and a set of parameters to have the general form `f(x, *f_args, **f_kwargs)`. That is, the parameters can either be positional or keyword arguments. The `diff` function must take the arguments `*f_args` and `**f_kwargs` and transfer these to `f`:

```
def diff(f, x, h=1E-10, *f_args, **f_kwargs):
    print f_args, f_kwargs
    return (f(x+h, *f_args, **f_kwargs) -
            f(x,   *f_args, **f_kwargs))/h
```

This `diff` function gives the writer of an `f` function full freedom to choose positional and/or keyword arguments for the parameters. Here is an example of the `G` function where we let the `t` parameter be positional and the other parameters be keyword arguments:

```
def G(x, t, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

A call

```
dGdx = diff(G, 0.5, 1E-9, 0, A=1, w=100, a=1.5)
```

gives the output

```
(0,) {'A': 1, 'a': 1.5, 'w': 100}
```

showing that `t` is put in `f_args` and transferred as positional argument to `G`, while `A`, `a`, and `w` are put in `f_kwargs` and transferred as keyword arguments. We remark that in the last call to `diff`, `h` and `t` *must* be treated as positional arguments, i.e., we cannot write `h=1E-9` and `t=0` unless *all* arguments in the call are on the `name=value` form.

In the case we use both `*f_args` and `**f_kwargs` arguments in `f` and there is no need for these arguments, `*f_args` becomes an empty tuple and `**f_kwargs` becomes an empty dictionary. The example

```
mycos = diff(sin, 0)
```

shows that the tuple and dictionary are indeed empty since `diff` just prints out

```
() {}
```

Therefore, a variable set of positional and keyword arguments can be incorporated in a general library function such as `diff` without any disadvantage, just the benefit that `diff` works with different types `f` functions: parameters as global variables, parameters as additional positional arguments, parameters as additional keyword arguments, or parameters as instance variables (Chapter 7.1.2).

The program `varargs1.py` in the `appendix` folder implements the examples in this appendix.

## E.6 Evaluating Program Efficiency

### E.6.1 Making Time Measurements

Time is not just "time" on a computer. The *elapsed time* or *wall clock time* is the same time as you can measure on a watch or wall clock, while *CPU time* is the amount of time the program keeps the central processing unit busy. The *system time* is the time spent on operating

system tasks like I/O. The concept *user time* is the difference between the CPU and system times. If your computer is occupied by many concurrent processes, the CPU time of your program might be very different from the elapsed time.

*The time Module.* Python has a `time` module with some useful functions for measuring the elapsed time and the CPU time:

```
import time
e0 = time.time()     # elapsed time since the epoch
c0 = time.clock()    # total CPU time spent in the program so far
<do tasks...>
elapsed_time = time.time() - e0
cpu_time = time.clock() - c0
```

The term *epoch* means initial time (`time.time()` would return 0), which is 00:00:00 January 1, 1970. The `time` module also has numerous functions for nice formatting of dates and time, and the more recent `datetime` module has more functionality and an improved interface. Although the timing has a finer resolution than seconds, one should construct test cases that last some seconds to obtain reliable results.

*The timeit Module.* To measure the efficiency of a certain set of statements or an expression, the code should be run a large number of times so the overall CPU-time is of order seconds. The `timeit` module has functionality for running a code segment repeatedly. Below is an illustration of `timeit` for comparing the efficiency `sin(1.2)` versus `math.sin(1.2)`:

```
>>> import timeit
>>> t = timeit.Timer('sin(1.2)', setup='from math import sin')
>>> t.timeit(10000000)   # run 'sin(1.2)' 10000000 times
11.830688953399658
>>> t = timeit.Timer('math.sin(1.2)', setup='import math')
>>> t.timeit(10000000)
16.234833955764771
```

The first argument to the `Timer` constructor is a string containing the code to execute repeatedly, while the second argument is the necessary code for initialization. From this simple test we see that `math.sin(1.2)` runs almost 40 percent slower than `sin(1.2)`!

If you want to time a function, say `f`, defined in the same program as where you have the `timeit` call, the setup procedure must import `f` and perhaps other variables from the program, as exemplified in

```
t = timeit.Timer('f(a,b)', setup='from __main__ import f, a, b')
```

Here, `f`, `a`, and `b` are names initialized in the main program. Another example is found in `src/random/smart_power.py`.

*Hardware Information.* Along with CPU-time measurements it is often convenient to print out information about the hardware on which the

experiment was done. Python has a module `platform` with information on the current hardware. The function `scitools.misc.hardware_info` applies the `platform` module to extract relevant hardware information. A sample call is

```
>>> import scitools.misc, pprint
>>> pprint.pprint(scitools.misc.hardware_info())
{'cpuinfo':
  {'CPU speed': '1196.170 Hz',
   'CPU type': 'Mobile Intel(R) Pentium(R) III CPU - M  1200MHz',
   'cache size': '512 KB',
   'vendor ID': 'GenuineIntel'},
 'identifier': 'Linux-2.6.12-i686-with-debian-testing-unstable',
 'python build': ('r25:409', 'Feb 27 2007 19:35:40'),
 'python version': '2.5.0',
 'uname': ('Linux',
           'ubuntu',
           '2.6.12',
           '#1 Fri Nov 25 10:58:24 CET 2005',
           'i686',
           '')}
```

### E.6.2 Profiling Python Programs

A profiler computes the time spent in the various functions of a program. From the timings a ranked list of the most time-consuming functions can be created. This is an indispensable tool for detecting bottlenecks in the code, and you should always perform a profiling before spending time on code optimization. The golden rule is to first write an easy-to-understand program, then verify it, then profile it, and then think about optimization[2].

Python comes with two profilers implemented in the `profile` and `hotshot` modules, respectively. The Python Library Reference has a good introduction to profiling in Python (Chapter 10: "The Python Profiler"). The results produced by the two alternative modules are normally processed by a special statistics utility `pstats` developed for analyzing profiling results. The usage of the `profile`, `hotshot`, and `pstats` modules is straightforward, but somewhat tedious so SciTools comes with a command `scitools profiler` that allows you to profile any program (say) `m.py` by writing

Terminal

```
Unix/DOS> scitools profiler m.py c1 c2 c3
```

Here, `c1`, `c2`, and `c3` are command-line arguments to `m.py`.

We refer to the Python Library Reference for detailed information on how to interpret the output. A sample output might read

[2] "Premature optimization is the root of all evil." –Donald Knuth, computer scientist, 1938-.

```
    1082 function calls (728 primitive calls) in 17.890 CPU seconds

Ordered by: internal time
List reduced from 210 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     5    5.850    1.170    5.850    1.170 m.py:43(loop1)
     1    2.590    2.590    2.590    2.590 m.py:26(empty)
     5    2.510    0.502    2.510    0.502 m.py:32(myfunc2)
     5    2.490    0.498    2.490    0.498 m.py:37(init)
     1    2.190    2.190    2.190    2.190 m.py:13(run1)
     6    0.050    0.008   17.720    2.953 funcs.py:126(timer)
...
```

In this test, `loop1` is the most expensive function, using 5.85 seconds, which is to be compared with 2.59 seconds for the next most time-consuming function, `empty`. The `tottime` entry is the total time spent in a specific function, while `cumtime` reflects the total time spent in the function and all the functions it calls.

The CPU time of a Python program typically increases with a factor of about five when run under the administration of the `profile` module. Nevertheless, the relative CPU time among the functions are probably not much affected by the profiler overhead.

# References


[1] D. Beazley. *Python Essential Reference*. SAMS, 2nd edition, 2001.

[2] J. E. Grayson. *Python and Tkinter Programming*. Manning, 2000.

[3] D. Harms and K. McDonald. *The Quick Python Book*. Manning, 1999.

[4] D. E. Knuth. Theory and practice. *EATCS Bull.*, 27:14–21, 1985.

[5] H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, third edition, 2009.

[6] L. S. Lerner. *Physics for Scientists and Engineers*. Jones and Barlett, 1996.

[7] M. Lutz. *Programming Python*. O'Reilly, second edition, 2001.

[8] M. Lutz and D. Ascher. *Learning Python*. O'Reilly, 1999.

[9] A. Martelli. *Python in a Nutshell*. O'Reilly, 2003.

[10] J. D. Murray. *Mathematical Biology I: An Introduction*. Springer, 3rd edition, 2007.

[11] F. M. White. *Fluid Mechanics*. McGraw-Hill, 2nd edition, 1986.

# Index